МИНИСТЕРСТВО ОБЩЕГО
И ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

П.В.Вельтмандер

МАШИННАЯ ГРАФИКА
(Учебное пособие в 3-х книгах)

Книга 2

ОСНОВНЫЕ АЛГОРИТМЫ

Новосибирск
1997

УДК 681.3.06
ББК В 185.2я73–1

Вельтмандер П.В. Машинная графика: Учеб. пособие в 3-х книгах. Книга 2. Основные алгоритмы/ Новосиб. ун-т. Новосибирск, 1997. 193 с., ил.

ISBN 5–230–13606–5

Учебное пособие представляет собой семестровый курс лекций. Содержит описание основных алгоритмов двух- трехмерной машинной графики, включая алгоритмы реалистичного представления сцен. Важную часть пособия составляют практические реализации алгоритмов на языке С. Как правило, приводится несколько реализаций для каждого алгоритма, отличающихся различным выбором между наглядностью и эффективностью.

Предназначено для обучения в вузах и колледжах, а также для специалистов — разработчиков программно-технических средств машинной графики и разработчиков прикладных пакетов, приближенных к техническим средствам.

Рецензент
канд. физ.-мат. наук С.И. Упольников

ISBN 5–230–13606–5

Оглавление

ВВЕДЕНИЕ

Данная, вторая часть курса лекций посвящена рассмотрению основных алгоритмов машинной графики.

В разделе 1 рассматриваются алгоритмы выполнения преобразований в двумерных, трехмерных и однородных координатах; параллельные, перспективные и стереопроекции; плоские преобразования растровых картин.

В разделе 2 рассматриваются три алгоритма генерации векторов — обычного и несимметричного ЦДА и Брезенхема. Там же рассмотрены способы борьбы с лестничным эффектом, вызванным различимыми размерами пикселов на экране. Один из способов основан на модификации алгоритма Брезенхема. Другой, общий способ базируется на использовании низкочастотной фильтрации. Этот способ, естественно, применим для произвольных изображений.

В разделе 3 приводится алгоритм генерации окружностей.

В разделе 4 рассмотрены различные алгоритмы заполнения многоугольника, заданного координатами его вершин. Там же рассмотрен наиболее быстрый алгоритм сортировки — алгоритм распределяющего подсчета.

В разделе 5 рассмотрены алгоритмы заливки с затравкой произвольной области, заданной либо значением граничных пикселов, либо значением пикселов внутренней части области.

Раздел 6 посвящен различным алгоритмам отсечения отрезка (Коэна-Сазерленда, Собкова-Поспишила-Янга, Лианга-Барски и Кируса-Бека) применительно к двух, трех и четырехмерным координатам.

В разделе 7 рассмотрены алгоритмы отсечения многоугольника.

В разделе 8 рассмотрены различные варианты организации данных.

В разделе 9 рассматривается геометрическое моделирование объектов и сцен.

Раздел 10 посвящен рассмотрению алгоритмов удаления скрытых линий и поверхностей.

В разделе 11 рассмотрены методы и алгоритмы реалистичного представления сцен.

В приложениях помещены процедуры на языке C, реализующие большую часть рассмотренные алгоритмы, а также тестовые программы для большинства процедур. Основной целью при написании процедур было достижение наглядности, поэтому есть возможности их оптимизации.

0.1 КООРДИНАТЫ И ПРЕОБРАЗОВАНИЯ

Описание, конструирование, манипулирование и представление геометрических объектов являются центральными работами в графических системах. Их поддержка в требуемом объеме за счет соответствующих математических методов, алгоритмов и программ оказывают существенное влияние на возможности и эффективность графической системы. В данном разделе будут рассмотрены математические методы для описания геометрических преобразований координат в двух, трех и четырехмерном случае, будут обсуждены некоторые вопросы эффективности, рассмотрены геометрические преобразования растровых картин.

Далее большими буквами X, Y, Z будут обозначаться обычные декартовые координаты, а маленькие буквы x, y, z будут использоваться для обозначения т.н. однородных координат.

0.1.1 Двумерные преобразования

в плоском случае имеет вид:

$$\mathrm{Xn} = \mathrm{X} + \mathrm{Tx}, \qquad \mathrm{Yn} = \mathrm{Y} + \mathrm{Ty}, \tag{0.1.1}$$

или в векторной форме:

$$\mathbf{Pn} = \mathbf{P} + \mathbf{T}, \tag{0.1.2}$$

где X, Y — исходные координаты точки,
Tx, Ty — величина сдвига по осям,
Xn, Yn — преобразованные координаты.
P = [X Y] — вектор-строка исходных координат,
Pn = [Xn Yn] — вектор-строка преобразованных координат,
T = [Tx Ty] — вектор-строка сдвига.

относительно начала координат имеет вид:

$$\mathrm{Xn} = \mathrm{X} \cdot \mathrm{Sx}, \qquad \mathrm{Yn} = \mathrm{Y} \cdot \mathrm{Sy}, \tag{0.1.3}$$

или в матричной форме:

$$\mathrm{Pn} = \mathrm{P} \cdot \mathrm{S}, \tag{0.1.4}$$

где Sx, Sy — коэффициенты масштабирования по осям, а

$$\mathrm{S} = \begin{bmatrix} \mathrm{Sx} & 0 \\ 0 & \mathrm{Sy} \end{bmatrix} \text{ — матрица масштабирования.}$$

относительно начала координат имеет вид:

$$\mathrm{Xn} = \mathrm{X} \cdot \cos\phi - \mathrm{Y} \cdot \sin\phi, \qquad \mathrm{Yn} = \mathrm{X} \cdot \sin\phi + \mathrm{Y} \cdot \cos\phi, \tag{0.1.5}$$

или в матричной форме:

$$\mathrm{Pn} = \mathrm{P} \cdot \mathrm{R}, \tag{0.1.6}$$

где ϕ — угол поворота, а

$$\mathrm{R} = \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} \text{ — матрица поворота.}$$

Столбцы и строки матрицы поворота представляют собой взаимно ортогональные единичные векторы. В самом деле квадраты длин векторов-строк равны единице:

$$\cos\phi \cdot \cos\phi + \sin\phi \cdot \sin\phi = 1$$

$$(-\sin\phi) \cdot (-\sin\phi) + \cos\phi \cdot \cos\phi = 1,$$

а скалярное произведение векторов-строк есть

$$\cos\phi \cdot (-\sin\phi) + \sin\phi \cdot \cos\phi = 0.$$

Так как скалярное произведение векторов $\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}| \cdot |\mathbf{B}| \cdot \cos\psi$, где $|\mathbf{A}|$ — длина вектора $\mathbf{A}$, $|\mathbf{B}|$ — длина вектора $\mathbf{B}$, а ψ — наименьший положительный угол между ними, то из равенства 0 скалярного произведения двух векторов-строк длины 1 следует, что угол между ними равен $90°$.

Аналогичное можно показать и для векторов-столбцов. Кроме того вектора-столбцы представляют собой такие единичные векторы, которые после выполнения преобразования, заданного этой матрицей, совпадут с осями. В самом деле, произведение первого столбца на матрицу есть

$$\begin{bmatrix} \cos\phi & -\sin\phi \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \tag{0.1.7}$$

т.е. это единичный вектор вдоль оси X. Аналогично, произведение второго столбца на матрицу даст вектор $[\,0\;1\,]$. Это позволяет сформировать матрицу, если известны результаты преобразования (см. пример в п. 0.1.5).

0.1.2 Двумерные преобразования в однородных координатах

Как видно из (0.1.2), (0.1.4) и (0.1.6) двумерные преобразования имеют различный вид. Сдвиг реализуется сложением, а масштабирование и поворот — умножением. Это различие затрудняет формирование суммарного преобразования и устраняется использованием двумерных однородных координат точки, имеющих вид:

$$[\,\mathrm{x}\;\mathrm{y}\;\mathrm{w}\,].$$

Здесь w — произвольный множитель не равный 0.

Двумерные декартовые координаты точки получаются из однородных делением на множитель w:

$$\mathrm{X} = \mathrm{x}/\mathrm{w}, \qquad \mathrm{Y} = \mathrm{y}/\mathrm{w}. \tag{0.1.8}$$

Однородные координаты можно представить как промасштабированные с коэффициентом w значения двумерных координат, расположенные в плоскости с $\mathrm{Z} = \mathrm{w}$.

В силу произвольности значения w в однородных координатах не существует единственного представления точки, заданной в декартовых координатах.

Преобразования сдвига, масштабирования и поворота в однородных координатах относительно центра координат все имеют одинаковую форму произведения вектора исходных координат на матрицу преобразования.

Для сдвига

$$\left[\begin{array}{ccc} \mathrm{xn} & \mathrm{yn} & \mathrm{wn} \end{array}\right] = \left[\begin{array}{ccc} \mathrm{x} & \mathrm{y} & \mathrm{w} \end{array}\right] \cdot \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathrm{Tx} & \mathrm{Ty} & 1 \end{array}\right]. \tag{0.1.9}$$

Для масштабирования

$$\left[\begin{array}{ccc} \mathrm{xn} & \mathrm{yn} & \mathrm{wn} \end{array}\right] = \left[\begin{array}{ccc} \mathrm{x} & \mathrm{y} & \mathrm{w} \end{array}\right] \cdot \left[\begin{array}{ccc} \mathrm{Sx} & 0 & 0 \\ 0 & \mathrm{Sy} & 0 \\ 0 & 0 & 1 \end{array}\right]. \tag{0.1.10}$$

Для поворота

$$\left[\begin{array}{ccc} \mathrm{xn} & \mathrm{yn} & \mathrm{wn} \end{array}\right] = \left[\begin{array}{ccc} \mathrm{x} & \mathrm{y} & \mathrm{w} \end{array}\right] \cdot \left[\begin{array}{ccc} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{array}\right]. \tag{0.1.11}$$

Как видно из (0.1.9) — (0.1.11), wn = w, а матрица преобразования для двумерных однородных координат в общем случае имеет вид:

$$\left[\begin{array}{ccc} \mathrm{A} & \mathrm{B} & \mathrm{P} \\ \mathrm{D} & \mathrm{E} & \mathrm{Q} \\ & & \\ \mathrm{L} & \mathrm{M} & \mathrm{S} \end{array}\right], \tag{0.1.12}$$

где элементы A, B, D и E определяют изменение масштаба, поворот и смещение, а L и M определяют сдвиг. Покажем, что элемент S определяет общее изменение масштаба, а элементы P и Q определяют проецирование.

Рассмотрим вначале для этого преобразование

$$\left[\begin{array}{ccc} \mathrm{xn} & \mathrm{yn} & \mathrm{h} \end{array}\right] = \left[\begin{array}{ccc} \mathrm{x} & \mathrm{y} & 1 \end{array}\right] \cdot \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & S \end{array}\right].$$

Легко видеть, что xn = x, yn = y, h = S. Таким образом двумерные декартовые координаты преобразованной точки

$$\mathrm{Xn} = \mathrm{xn}/\mathrm{h} = \mathrm{x}/\mathrm{S}, \qquad \mathrm{Yn} = \mathrm{yn}/\mathrm{h} = \mathrm{y}/\mathrm{S},$$

т.е. такое преобразование задает изменение масштаба вектора положения точки. При S < 1 выполняется уменьшение, а при S > 1 — увеличение.

Для уяснения смысла третьего столбца матрицы преобразований (0.1.12) выполним преобразование

$$\left[\begin{array}{ccc} \mathrm{xn} & \mathrm{yn} & \mathrm{h} \end{array}\right] = \left[\begin{array}{ccc} \mathrm{x} & \mathrm{y} & 1 \end{array}\right] \cdot \left[\begin{array}{ccc} 1 & 0 & P \\ 0 & 1 & Q \\ 0 & 0 & 1 \end{array}\right] = \left[\begin{array}{ccc} \mathrm{x} & \mathrm{y} & (\mathrm{Px} + \mathrm{Qy} + 1) \end{array}\right].$$

Здесь xn = x, yn = y, h = Px + Qy + 1, т.е. переменная h, которая определяет плоскость, содержащую преобразованные точки, представленные в однородных координатах, образует теперь уравнение плоскости в трехмерном пространстве:

$$\mathrm{h} = \mathrm{Px} + \mathrm{Qy} + 1. \tag{0.1.13}$$

Получим результирующие двумерные декартовые координаты Xn, Yn для преобразованной точки

$$Xn = \frac{x}{Px + Qy + 1}, \qquad Yn = \frac{y}{Px + Qy + 1}.$$

Это соответствует вычислению их в плоскости Z = 1, т.е. проецированию из плоскости (0.1.13) в плоскость Z = 1. Легко показать, что центр проецирования находится в начале координат. Рассмотрим для этого параметрические уравнения прямой, проходящей через точки (X_0, Y_0, 1) и (X, Y, (MX + NY + 1)):

$$\begin{array}{rcrcrcrcl} X(t) = & X_0 & + & (X - X_0) \times t & = & x/h & + & (x - x/h) \times t \\ Y(t) = & Y_0 & + & (Y - Y_0) \times t & = & y/h & + & (y - y/h) \times t \\ Z(t) = & 1 & + & (MX + NY) \times t & = & 1 & + & (h - 1) \times t \end{array} . \qquad (0.1.14)$$

Из условия $X(t) = X_0 = 0$ находим $t = 1/(1 - h)$, подставляя это значение t в выражения для Y(t) и Z(t), получим:

$$Y_0 = y/h + (y - y/h)/(1 - h) = y/h - y/h = 0.$$

$$Z_0 = 1 + (h - 1)/(1 - h) = 0.$$

Итак, показано, что элементы P и Q матрицы (0.1.12) определяют проецирование с центром проекции в начале координат.

Кроме удобств, связанных с единообразным представлением преобразований, и, следовательно, с упрощением композиции преобразований, рассматриваемой в следующем пункте, однородные координаты дают возможность простого представления точек, имеющих в декартовой системе значение координаты, равное бесконечности.

Декартовые точки с бесконечными координатами

Рассмотрим в декартовой системе линию, проходящую через начало координат и точку (X, Y). Однородные координаты этой точки — (x, y, h) = (hX, hY, h), где h имеет произвольное значение. Предел отношения x/y при h стремящимся к 0 равен X/Y, но при этом декартовые координаты стремятся к бесконечности. Таким образом, точка с однородными координатами

$$(x,\ y,\ 0) \qquad (0.1.15)$$

задает в декартовой системе точку на бесконечности для рассмотренной прямой. В частности, точка с однородными координатами (1, 0, 0) задает бесконечную точку на декартовой оси X, а точка с однородными координатами (0, 1, 0) задает бесконечную точку на декартовой оси Y.

Параллельные прямые

Покажем, что прямые, параллельные в декартовой системе координат, в однородных координатах имеют точку пересечения. Эта особенность далее будет использована при анализе перспективных преобразований.

Пусть две пересекающиеся прямые в декартовой системе координат заданы системой уравнений:

$$\begin{array}{l} A_1 \cdot X + B_1 \cdot Y + C_1 = 0 \\ A_2 \cdot X + B_2 \cdot Y + C_2 = 0. \end{array} \qquad (0.1.16)$$

Решая эту систему относительно X и Y, найдем координаты точки пересечения

$$X_0 = \frac{C_1 \cdot B_2 - C_2 \cdot B_1}{A_1 \cdot B_2 - A_2 \cdot B_1}.$$

$$Y_0 = \frac{A_1 \cdot C_2 - A_2 \cdot C_1}{A_1 \cdot B_2 - A_2 \cdot B_1}.$$

Запишем результат в однородных координатах

$$\left(\frac{C_1 \cdot B_2 - C_2 \cdot B_1}{A_1 \cdot B_2 - A_2 \cdot B_1},\ \frac{A_1 \cdot C_2 - A_2 \cdot C_1}{A_1 \cdot B_2 - A_2 \cdot B_1},\ 1\right).$$

В силу произвольности масштабного множителя, умножим значения координат на $(A_1 \cdot B_2 - A_2 \cdot B_1)$

$$(C_1 \cdot B_2 - C_2 \cdot B_1,\ A_1 \cdot C_2 - A_2 \cdot C_1,\ A_1 \cdot B_2 - A_2 \cdot B_1).$$

Если прямые параллельны, то определитель системы (0.1.16) — $(A_1 \cdot B_2 - A_2 \cdot B_1)$ равен нулю. Учитывая это и обозначая $x_0 = (C_1 \cdot B_2 - C_2 \cdot B_1)$, $y_0 = (A_1 \cdot C_2 - A_2 \cdot C_1)$, получим координату пересечения параллельных прямых в однородной системе координат

$$(\ x_0,\ y_0,\ 0\ ).$$

При этом точка пересечения лежит на прямой $-y_0 \cdot x + x_0 \cdot y = 0$ на бесконечности.

0.1.3 Композиция двумерных преобразований

Последовательное выполнение нескольких преобразований можно представить в виде единой матрицы суммарного преобразования. Умножение на единственную матрицу, естественно, выполняется быстрее, чем последовательное умножение на несколько матриц.

Рассмотрим сдвиг точки P_0 на расстояние (Tx_1, Ty_1) в точку P_1, а затем сдвинем точку P_1 на расстояние (Tx_2, Ty_2) в точку P_2. Обозначая через T_1 и T_2 матрицы сдвига, в соответствии с (0.1.9) получим:

$$P_1 = P_0 \cdot T_1;\ P_2 = P_1 \cdot T_2\ =\ (P_0 \cdot T_1) \cdot T_2\ =\ P_0 \cdot (T_1 \cdot T_2)\ =\ P_0 \cdot T.$$

Понятно, что сдвиг аддитивен, т.е. последовательное выполнение двух сдвигов должно быть эквивалентно одному сдвигу на расстояние $(Tx_1 + Tx_2, Ty_1 + Ty_2)$. Для доказательства этого рассмотрим произведение матриц сдвига T_1 и T_2, равное

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx_1 & Ty_1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx_2 & Ty_2 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Tx_1 + Tx_2 & Ty_1 + Ty_2 & 1 \end{bmatrix}.$$

Итак, получили, что результирующий сдвиг есть $(Tx_1 + Tx_2,\ Ty_1 + Ty_2)$, т.е. суммарный сдвиг, вычисленный как произведение матриц, как и ожидалось, аддитивен.

Рассмотрим теперь последовательное выполнение масштабирований, первое с коэффициентами (Sx_1, Sy_1), второе с коэффициентами (Sx_2, Sy_2). Следует ожидать, что суммарное масштабирование будет мультипликативным. Обозначая через S_1 и S_2 матрицы масштабирования, в соответствии с (0.1.10) получим

$$P_1 = P_0 \cdot S_1,\ P_2 = P_1 \cdot S_2 \ = \ (P_0 \cdot S_1) \cdot S_2 \ = \ P_0 \cdot (S_1 \cdot S_2) = P_0 \cdot S.$$

Найдем значения элементов матрицы S

$$S = \begin{bmatrix} Sx_1 & 0 & 0 \\ 0 & Sy_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} Sx_2 & 0 & 0 \\ 0 & Sy_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Sx_1 \cdot Sx_2 & 0 & 0 \\ 0 & Sy_1 \cdot Sy_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Итак, получили, что результирующее масштабирование есть ($Sx_1 \cdot Sx_2$, $Sy_1 \cdot Sy_2$), т.е. суммарное масштабирование, вычисленное как произведение матриц, как и ожидалось, мультипликативно.

Аналогичным образом можно показать, что два последовательных поворота аддитивны.

Рассмотрим выполнение часто используемого поворота изображения на угол ϕ относительно заданной точки P(X, Y). Это преобразование можно представить как перенос начала координат в точку (X, Y), поворот на угол ϕ относительно начала координат и обратный перенос начала координат:

$$Pn = P \cdot T(-X, -Y) \cdot R(\phi) \cdot T(X, Y).$$

С использованием преобразований в однородных координатах, суммарное преобразование будет иметь простой вид:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -X & -Y & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ X & Y & 1 \end{bmatrix}.$$

0.1.4 Эффективность преобразований

Суммарная матрица двумерных преобразований в однородных координатах имеет вид:

$$\begin{bmatrix} A & B & 0 \\ D & E & 0 \\ L & M & 1 \end{bmatrix},$$

где элементы A, B, D и E, отвечающие за изменение масштаба, поворот и смещение, — объединенная матрица масштабирования и поворота, а L и M определяют суммарный сдвиг.

Вычисление преобразованных однородных координат точки P с непосредственным использованием T в выражении $P \cdot T$ требует 9 операций умножения и 6 операций сложения. Но так как третья однородная координата может быть выбрана равной 1, а третий столбец T содержит единственный ненулевой элемент, равный 1, то преобразование декартовых координат может быть представлено в виде:

$$Xn = X \cdot A + Y \cdot D + L, \qquad Yn = X \cdot B + Y \cdot E + M,$$

что требует уже только 4 операции умножения и 4 операции сложения, что существенно меньше. Таким образом, несмотря на то, что матрицы 3×3 удобны при вычислении суммарного

преобразования, выполнение фактического преобразования координат следует производить с учетом реальной структуры матрицы преобразования.

0.1.5 Трехмерные координаты

Далее при рассмотрении трехмерных преобразований, в основном, используется общепринятая в векторной алгебре правая система координат (рис. 0.1.1а). При этом, если смотреть со стороны положительной полуоси в центр координат, то поворот на $+90^\circ$ (против часовой стрелке) переводит одну положительную ось в другую (направление движения расположенного вдоль оси и поворачивающегося против часовой стрелки правого винта и положительной полуоси совпадают). В некоторых, специально оговариваемых случаях, используется левая система координат (см. рис. 0.1.1б). В левой системе координат положительными будут повороты по часовой стрелке, если смотреть с положительного конца полуоси. В трехмерной машинной графике более удобной является левая система координат. Тогда если, например, поверхность экрана совмещена с плоскостью XY, то большим удалениям от наблюдателя соответствую точки с большим значением Z (см. рис. 0.1.1б). Работа с однородными трехмерными координатами

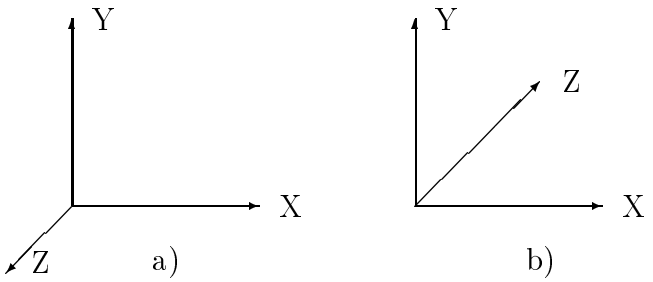


Рис. 0.1.1: Правая а) и левая б) системы координат

и матрицами преобразования (формирование и композиция) подобна таковой для двумерного случая, поэтому здесь будут рассмотрены только матрицы преобразований сдвига, масштабирования и поворота и пример конструирования матрицы преобразования по известному его результату.

Подобно тому как в двумерном случае точка в однородных координатах представляется трехмерным вектором $[\,\mathrm{x\,y\,w}\,]$, а матрицы преобразований имеют размер 3×3, для трехмерного случая точка представляется четырехмерным вектором $[\,\mathrm{x\ y\ z\ w}\,]$, где w не равно 0, а матрицы преобразований имеют размер 4×4. Если w не равно 1, то декартовые координаты точки $(\mathrm{X, Y, Z})$ получаются из соотношения:

$$[\,\mathrm{X\ Y\ Z\ 1}\,] = [\,(\mathrm{x/w})\ (\mathrm{y/w})\ (\mathrm{z/w})\ 1\,].$$

Преобразование в однородных координатах описывается соотношением

$$[\,\mathrm{xn\ yn\ zn\ wn}\,] = [\,\mathrm{x\ y\ z\ w}\,] \cdot \mathrm{T}.$$

Матрица преобразования T в общем случае имеет вид

$$\begin{bmatrix} \mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{P} \\ \mathrm{D} & \mathrm{E} & \mathrm{F} & \mathrm{Q} \\ \mathrm{I} & \mathrm{J} & \mathrm{K} & \mathrm{R} \\ \mathrm{L} & \mathrm{M} & \mathrm{N} & \mathrm{S} \end{bmatrix}.$$

Подматрица 3×3 определяет суммарные смещение, масштабирование и поворот. Подматрица-строка 1×3 — [L M N] задает сдвиг. Подматрица-столбец 3×1 — [P Q R] отвечает за преобразование в перспективе. Последний скалярный элемент — S определяет общее изменение масштаба.

В частности, матрица сдвига имеет вид:

$$\mathrm{T}(\mathrm{Tx}, \mathrm{Ty}, \mathrm{Tz}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathrm{Tx} & \mathrm{Ty} & \mathrm{Tz} & 1 \end{bmatrix}.$$

Матрица обратного преобразования для сдвига получается путем смены знака у Tx, Ty и Tz.

Матрица масштабирования относительно центра координат имеет вид:

$$\mathrm{S}(\mathrm{Sx}, \mathrm{Sy}, \mathrm{Sz}) = \begin{bmatrix} \mathrm{Sx} & 0 & 0 & 0 \\ 0 & \mathrm{Sy} & 0 & 0 \\ 0 & 0 & \mathrm{Sz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матрица обратного преобразования для масштабирования формируется при замене Sx, Sy и Sz на величины, обратные к ним.

Ранее рассмотренная для двумерного случая матрица поворота (0.1.11) является в то же время трехмерным поворотом вокруг оси Z. Так как при трехмерном повороте вокруг оси Z (поворот в плоскости XY) размеры вдоль оси Z неизменны, то все элементы третьей строки и третьего столбца равны 0, кроме диагонального, равного 1:

$$\mathrm{Rz}(\phi_{\mathrm{z}}) = \begin{bmatrix} \cos\phi_z & \sin\phi_z & 0 & 0 \\ -\sin\phi_z & \cos\phi_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

При повороте вокруг оси X (в плоскости YZ) размеры вдоль оси X не меняются, поэтому все элементы первой строки и первого столбца равны 0, за исключением диагонального, равного 1:

$$\mathrm{Rx}(\phi_{\mathrm{x}}) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi_x & \sin\phi_x & 0 \\ 0 & -\sin\phi_x & \cos\phi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

При повороте вокруг оси Y (в плоскости XZ) размеры вдоль оси Y не меняются, поэтому все элементы второй строки и второго столбца равны 0, за исключением диагонального, равного 1:

$$\mathrm{Ry}(\phi_{\mathrm{y}}) = \begin{bmatrix} \cos\phi_y & 0 & -\sin\phi_y & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi_y & 0 & \cos\phi_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Столбцы и строки подматриц 3×3 матриц поворота Rx, Ry, Rz, аналогично двумерному случаю, представляют собой взаимно ортогональные единичные векторы. Легко убедиться, что

суммарная матрица преобразования для произвольной последовательности поворотов вокруг осей X, Y и Z имеет вид:

$$R = \begin{bmatrix} r1_x & r1_y & r1_z & 0 \\ r2_x & r2_y & r2_z & 0 \\ r3_x & r3_y & r3_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

причем столбцы (и строки) представляют собой взаимно ортогональные единичные векторы. Более того, векторы-столбцы при повороте, задаваемом матрицей, совмещаются с соответствующими осями координат. Матрица, столбцы (или строки) которой представляют собой взаимно ортогональные векторы, называется ортогональной. Для любой ортогональной матрицы обратная матрица совпадает с транспонированной. Это обеспечивает простоту вычисления обратного преобразования для поворота. Причем не надо фактически выполнять транспонирование, а достаточно просто поменять местами индексы строк и столбцов.

Взаимная ортогональность столбцов матрицы поворота и их совмещение с осями координат при преобразовании, задаваемом матрицей, позволяет легко сконструировать матрицу преобразования, если известны его результаты.

Пример формирования матрицы преобразования (из [4])

Пусть заданы три точки P_1, P_2, P_3. Найти матрицу преобразования такого, что после преобразования вектор $\mathbf{P_1P_2}$ будет направлен вдоль оси Z, а вектор $\mathbf{P_1P_3}$ будет лежать в плоскости YZ.

1. Вначале надо сместить начало координат в точку P_1 с помощью преобразования

$$T(-x_1, -y_1, -z_1).$$

2. Единичный вектор, который должен лечь вдоль оси Z

$$\mathbf{R}_z = [r1_z\ r2_z\ r3_z] = \mathbf{P}_1\mathbf{P}_2/|\mathbf{P}_1\mathbf{P}_2|.$$

Здесь $|\mathbf{P}_1\mathbf{P}_2|$ — длина вектора $\mathbf{P}_1\mathbf{P}_2$.

3. Вектор, перпендикулярный плоскости, построенной на векторах $\mathbf{P}_1\mathbf{P}_2$ и $\mathbf{P}_1\mathbf{P}_3$, должен быть направлен вдоль оси X, так как вектор $\mathbf{P}_1\mathbf{P}_2$ лежит вдоль оси Z, а вектор $\mathbf{P}_1\mathbf{P}_3$ лежит в плоскости YZ. Этот вектор задается векторным произведением

$$\mathbf{R}_x = [r1_x\ r2_x\ r3_x] = \frac{\mathbf{P}_1\mathbf{P}_2 \times \mathbf{P}_1\mathbf{P}_3}{|\mathbf{P}_1\mathbf{P}_2| \cdot |\mathbf{P}_1\mathbf{P}_3|}.$$

4. Наконец, вдоль оси Y должен быть направлен вектор, перпендикулярный к векторам $\mathbf{R}_x\mathbf{R}_z$:

$$\mathbf{R}_y = [\ r1_y\ r2_y\ r3_y\ ] = \mathbf{R}_z \times \mathbf{R}_x.$$

5. Искомая матрица есть

$$M = T(-x_1, -y_1, -z_1) \cdot \begin{bmatrix} r1_x & r1_y & r1_z & 0 \\ r2_x & r2_y & r2_z & 0 \\ r3_x & r3_y & r3_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

0.1.6 Проекции

При визуализации двумерных изображений достаточно задать окно видимости в системе координат пользователя и порт отображения на экране дисплея, в котором будет выдаваться изображение из окна. В этом случае достаточно провести отсечение изображения по окну и выполнить двумерные преобразования окно-порт. На рис. 0.1.2 показан простой пример двумерных преобразований. Окном отсекается часть изображения домика и один улей (см. рис. 0.1.2 слева). Отсеченное изображение передается в порт отображения дисплея с выполнением преобразований окно-порт (см. рис. 0.1.2 справа). В данном (простом) случае выполняется только преобразование сдвига.

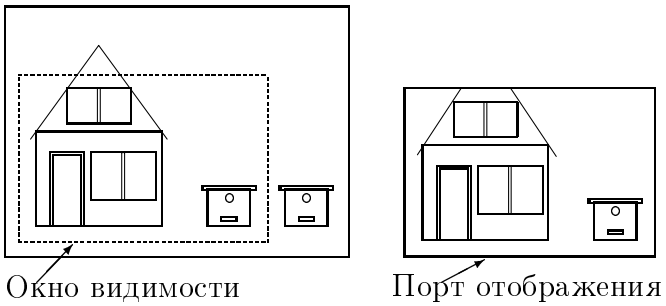


Рис. 0.1.2: Пример визуализации для двумерных изображений

В случае же трехмерных изображений отсечение выполняется уже не по окну, а по объему видимости и затем выполняется проецирование в порт отображения, который в свою очередь может быть проекцией объема видимости. Модель процесса визуализации трехмерных изображений приведена на рис. 0.1.3.

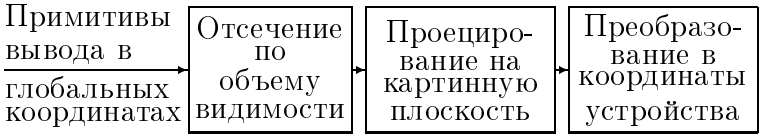


Рис. 0.1.3: Модель процесса визуализации трехмерных изображений

Как уже отмечалось, проецирование в общем случае — отображение точек, заданных в системе координат размерностью N, в точки в системе с меньшей размерностью. При отображении трехмерных изображений на дисплей три измерения отображаются в два.

Проецирование выполняется с помощью прямолинейных проекторов (проецирующих лучей), идущих из центра проекции через каждую точку объекта до пересечения с картинной поверхностью (поверхностью проекции). Далее рассматриваются только плоские проекции, при которых поверхность проекции — плоскость в трехмерном пространстве.

По расположению центра проекции относительно плоскости проекции различаются центральная и параллельные проекции.

При **параллельной проекции** центр проекции находится на бесконечном расстоянии от плоскости проекции. Проекторы представляют собой пучок параллельных лучей. В этом случае необходимо задавать направление проецирования и расположение плоскости проекции. По

взаимному расположению проекторов, плоскости проекции и главных осей координат различаются и
проекции.

При проекторы перпендикулярны плоскости проекции, а плоскость проекции перпендикулярна главной оси. Т.е. проекторы параллельны главной оси.
имеется одна из двух перпендикулярностей:

- проекторы перпендикулярны плоскости проекции, которая расположена под углом к главной оси;
- проекторы не перпендикулярны плоскости проекции, но плоскость проекции перпендикулярна к главной оси.

Изображение, полученное при параллельном проецировании, не достаточно реалистично, но передаются точные форма и размеры, хотя и возможно различное укорачивание для различных осей.

При **центральной проекции** расстояние от центра проекции до плоскости проецирования конечно, поэтому проекторы представляют собой пучок лучей, исходящих из центра проекции. В этом случае надо задавать расположение и центра проекции и плоскости проекции. Изображения на плоскости проекции имеют т.н. перспективные искажения, когда размер видимого изображения зависит от взаимного расположения центра проекции, объекта и плоскости проекции. Из-за перспективных искажений изображения, полученные центральной проекцией, более реалистичны, но нельзя точно передать форму и размеры. Различаются одно, двух и трехточечные центральные проекции в зависимости от того по скольким осям выполняется перспективное искажение. Иллюстрация центральной проекции приведена на рис. 0.1.12.

На рис. 0.1.4 приведена классификация описанных выше плоских проекций.

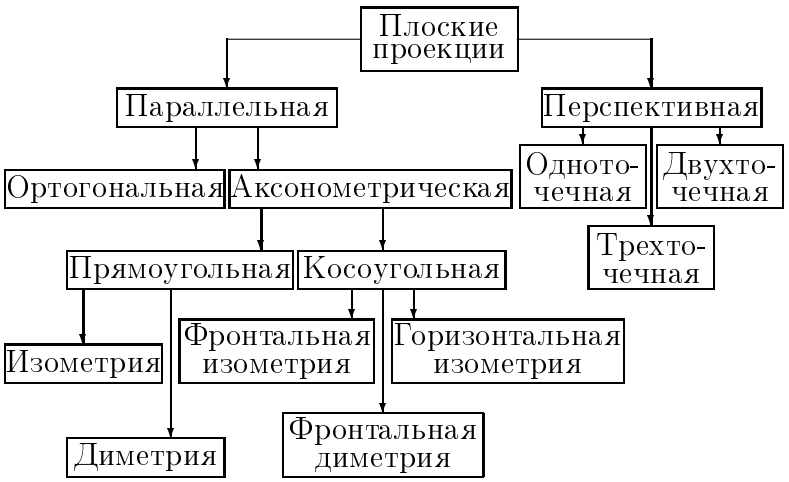


Рис. 0.1.4: Классификация плоских проекций

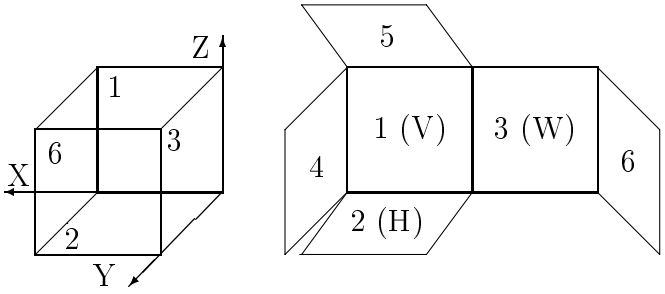
Параллельные проекции

Вначале мы рассмотрим ортогональные проекции, используемые в техническом черчении, в регламентированной для него правосторонней системе координат, когда ось Z изображается

вертикальной. Затем будут проиллюстрированы аксонометрические проекции также в правосторонней системе координат, но уже более близкой к машинной графике (ось Y вертикальна, ось X направлена горизонтально вправо, а ось Z — от экрана к наблюдателю). Наконец выведем матрицы преобразования в левосторонней системе координат, часто используемой в машинной графике, с вертикальной осью Y, осью X, направленной вправо и осью Z, направленной от наблюдателя.

Использование проекций в техническом черчении регламентируется стандартом ГОСТ 2.317–69. Наиболее широко, особенно, в САПР используются ортогональные проекции (виды). Вид — ортогональная проекция обращенной к наблюдателю видимой части поверхности предмета, расположенного между наблюдателем и плоскостью чертежа.

В техническом черчении за основные плоскости проекций принимают шесть граней куба (рис. 0.1.5).



1. Вид спереди, главный вид, фронтальная проекция, (на заднюю грань V),
2. Вид сверху, план, горизонтальная проекция, (на нижнюю грань H),
3. Вид слева, профильная проекция, (на правую грань W),
4. Вид справа (на левую грань),
5. Вид снизу (на верхнюю грань),
6. Вид сзади (на переднюю грань).

Рис. 0.1.5: Ортогональные проекции (основные виды) и их расположение на листе чертежа

Очевидно, что при ортогональной проекции не происходит изменения ни углов, ни масштабов.

При аксонометрическом проецировании (см. рис. 0.1.4) сохраняется параллельность прямых, а углы изменяются; измерение же расстояний вдоль каждой из координатных осей в общем случае должно выполняться со своим масштабным коэффициентом.

При изометрических проекциях укорачивания вдоль всех координатных осей одинаковы, поэтому можно производить измерения вдоль направлений осей с одним и тем же масштабом (отсюда и название изометрия). На рис. 0.1.6 приведена (аксонометрическая прямоугольная) изометрическая проекция куба со стороной A. При этой проекции плоскость проецирования наклонена ко всем главным координатным осям под одинаковым углом. Стандартом регламентируется коэффициент сжатия, равный 0.82, а также расположение и взаимные углы главных координатных осей, равные 120° как это показано в левом верхнем углу рис. 0.1.6. Обычно сжатие не делается.

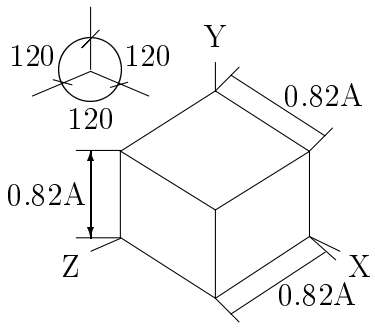


Рис. 0.1.6: Аксонометрическая прямоугольная изометрическая проекция куба со стороной A

При диметрической проекции две из трех осей сокращены одинаково, т.е. из трех углов между нормалью к плоскости проекции и главными координатными осями два угла одинаковы. На рис. 0.1.7 приведена (аксонометрическая прямоугольная) диметрическая проекция куба со стороной A. Там же показаны регламентируемые расположение осей и коэффициенты сжатия. Обычно вместо коэффициента сжатия 0.94 используется 1, а вместо 0.47 — 0.5.

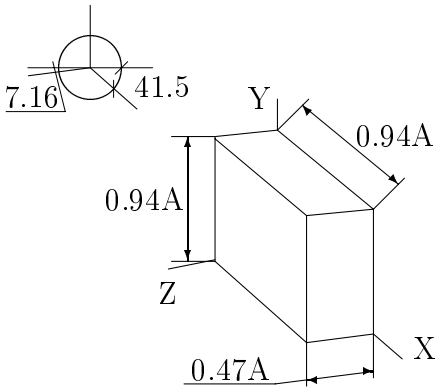


Рис. 0.1.7: Аксонометрическая прямоугольная диметрическая проекция куба со стороной A

В косоугольных проекциях плоскость проекции перпендикулярна главной координатной оси, а проекторы расположены под углом к ней. Таким образом, аксонометрические косоугольные проекции сочетают в себе свойства ортогональных и аксонометрических прямоугольных проекций.

Наиболее употребимы два вида косоугольной проекции — фронтальная (косоугольная) диметрия (проекция Kabinett — кабине) и горизонтальная (косоугольная) изометрия (проекция Kavalier — кавалье) или военная перспектива.

В случае фронтальной (косоугольной) диметрии при использовании правосторонней системы координат экрана плоскость проецирования перпендикулярна оси Z. Ось X направлена горизонтально вправо. Ось Z изображается по углом в 45° относительно горизонтального направления. Допускается угол наклона в 30 и 60°. При этом отрезки, перпендикулярные плоскости проекции, при проецирования сокращаются до 1/2 их истинной длины. На рис. 0.1.8 приведена (аксонометрическая косоугольная) фронтальная диметрическая проекция куба со стороной A, там же показаны регламентируемые коэффициент сжатия, равный 0.5 и расположение осей.

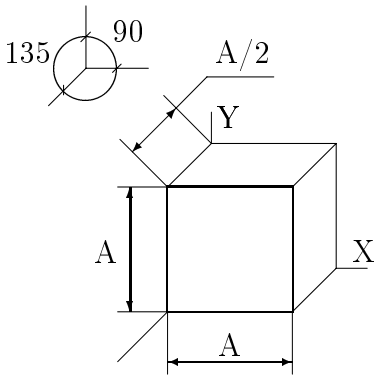


Рис. 0.1.8: Аксонометрическая косоугольная фронтальная диметрическая проекция куба со стороной A

В случае же (аксонометрической косоугольной) горизонтальной изометрии, как следует из названия, плоскость проецирования перпендикулярна оси Y а укорачивания по всем осям одинаковы и равны 1. Угол поворота изображения оси X относительно горизонтального направления составляет 30°. Допускается 45 и 60° при сохранении угла 90° между изображениями осей X и Z. Иллюстрация этого приведена на рис. 0.1.9.

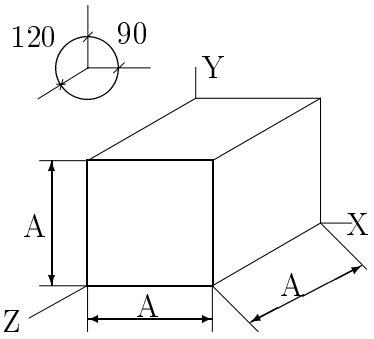


Рис. 0.1.9: Аксонометрическая косоугольная горизонтальная изометрическая проекция куба со стороной A

Выведем выражения для матриц преобразования, используя теперь левостороннюю систему координат более естественную для машинной графики.

Простейшее параллельное проецирование — ортогональное выполняется на плоскость, перпендикулярную какой-либо оси, т.е. при направлении проецирования вдоль этой оси. В частности, проецирование в XY-плоскость, заданную соотношением $Z = Z_0$, выполняется следующим образом:

$$\begin{bmatrix} xn & yn & zn & wn \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & Z_0 & 1 \end{bmatrix}.$$

Рассмотрим теперь косоугольное проецирование, при котором плоскость проецирования перпендикулярна главной оси, а проекторы составляют с плоскостью проецирования угол не рав-

ный 90°. Матрица для этого преобразования может быть найдена исходя из значений угла проецирования и координат преобразованной точки. На рис. 0.1.10 показана косоугольная параллельная проекция единичного куба.

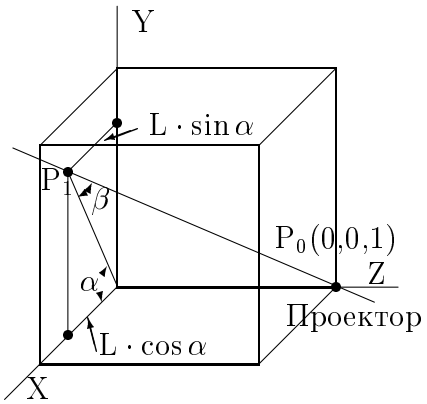


Рис. 0.1.10: Косоугольная параллельная проекция $\mathrm{P}_1(\mathrm{L}\cdot\cos\alpha, \mathrm{L}\cdot\sin\alpha, 0)$ точки $\mathrm{P}_0(0,0,1)$

Из рисунка видно, что проектором, идущим из точки P_0 в P_1, точка $\mathrm{P}_0(0,0,1)$ проецируется в $\mathrm{P}_1(\mathrm{L}\cdot\cos\alpha, \mathrm{L}\cdot\sin\alpha, 0)$.

Теперь проектором, параллельным рассмотренному (рис. 0.1.11), спроецируем некоторую точку (X,Y,Z) в точку (X_p, Y_p, Z_p).

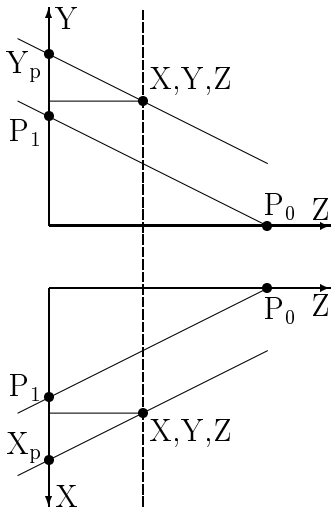


Рис. 0.1.11: Косоугольная параллельная проекция $(\mathrm{X_p}, \mathrm{Y_p}, 0)$ точки $(\mathrm{X}, \mathrm{Y}, \mathrm{Z})$

Из подобия треугольников получаем:

$$(\mathrm{X_p} - \mathrm{X})/\mathrm{Z} = \mathrm{L}\cdot\cos\alpha \quad \Rightarrow \quad \mathrm{Xp} = \mathrm{X} + \mathrm{Z}\cdot\mathrm{L}\cdot\cos\alpha$$

$$(\mathrm{Y_p} - \mathrm{Y})/\mathrm{Z} = \mathrm{L}\cdot\sin\alpha \quad \Rightarrow \quad \mathrm{Y_p} = \mathrm{Y} + \mathrm{Z}\cdot\mathrm{L}\cdot\sin\alpha$$

Это соответствует следующему матричному выражению:

$$\left[\begin{array}{cccc} \mathrm{xp} & \mathrm{yp} & \mathrm{zp} & 1 \end{array}\right] = \left[\begin{array}{cccc} \mathrm{x} & \mathrm{y} & \mathrm{z} & 1 \end{array}\right] \cdot \left[\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \mathrm{L}\cdot\cos\alpha & \mathrm{L}\cdot\sin\alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right].$$

Таким образом, матрица аксонометрической косоугольной проекции для случая проецирования в плоскость $\mathrm{Z}=0$, выполняет следующее:

- вначале плоскости с заданной координатой Z_0 переносятся вдоль оси X на $\mathrm{Z}_0\cdot\mathrm{L}\cdot\cos\alpha$ и вдоль оси Y на $\mathrm{Z}_0\cdot\mathrm{L}\cdot\sin\alpha$,
- затем производится проецирование в плоскость $\mathrm{Z}=0$.

Различные варианты параллельных проекций формируются из полученной подстановкой значений L и углов α и β (см. рис. 0.1.10). В частности, для фронтальной косоугольной диметрии $\mathrm{L}=1/2$, следовательно, угол β между проекторами и плоскостью проецирования равен $\arctan 2 = 63.4°$. Угол же α, равен 45° и допускается 30 и 60°, как это сказано выше. (Обратите внимание, что в этой системе координат плоскость фронтальной проекции — плоскость XY, в отличие от системы координат технического черчения, где фронтальная проекция, как это показано на рис. 0.1.5, формируется в плоскости XZ).

Центральная проекция

Наиболее реалистично трехмерные объекты выглядят в центральной проекции из-за перспективных искажений сцены. Центральные проекции параллельных прямых, не параллельных плоскости проекции будут сходиться в . В зависимости от числа точек схода, т.е. от числа координатных осей, которые пересекает плоскость проекции, различаются одно, двух и трехточечные центральные проекции. Иллюстрация одно-, двух- и трехточечной центральных проекций куба приведена на рис. 0.1.12.

Наиболее широко используется двухточечная центральная проекция.

Выведем матрицу, определяющую центральное проецирование для простого случая одноточечной проекции (рис. 0.1.13), когда плоскость проекции перпендикулярна оси Z и расположена на расстоянии d от начала координат. (Здесь используется удобная для машинной графики левосторонняя система координат).

Начало отсчета находится в точке просмотра. Ясно, что изображения объектов, находящиеся между началом координат и плоскостью проекции увеличиваются, а изображения объектов, расположенных дальше от начала координат, чем плоскость проекции уменьшаются.

Из рис. 0.1.13 видно, что для координат (X1,Y1) точки P_1, полученной проецированием точки $\mathrm{P}_0(\mathrm{X},\mathrm{Y},\mathrm{Z})$ в плоскость $\mathrm{Z}=\mathrm{d}$ (плоскость экрана) выполняются следующие соотношения:

$$\frac{\mathrm{X}_1}{\mathrm{d}} = \frac{\mathrm{X}}{\mathrm{Z}}, \qquad \frac{\mathrm{X}_1}{\mathrm{d}} = \frac{\mathrm{X}}{\mathrm{Z}}, \qquad \mathrm{X}_1 = \frac{\mathrm{X}}{\mathrm{Z}/\mathrm{d}}, \qquad \mathrm{Y}_1 = \frac{\mathrm{Y}}{\mathrm{Z}/\mathrm{d}}.$$

Такое преобразование может быть представлено матрицей 4×4

$$\left[\begin{array}{cccc} \mathrm{x}_1 & \mathrm{y}_1 & \mathrm{z}_1 & \mathrm{w}_1 \end{array}\right] = \left[\begin{array}{cccc} \mathrm{x} & \mathrm{y} & z & 1 \end{array}\right] \cdot \left[\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1/\mathrm{d} \\ 0 & 0 & 0 & 0 \end{array}\right] =$$

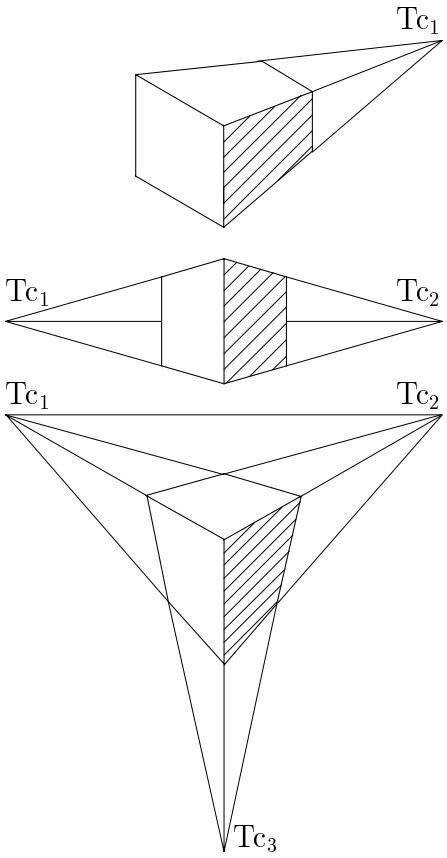


Рис. 0.1.12: Одно-, двух- и трехточечная центральные проекции куба

$$\left[\begin{array}{cccc} \mathrm{x} & \mathrm{y} & \mathrm{z} & 1 \end{array}\right] \cdot M_{\text{ц}} = \left[\begin{array}{cccc} \mathrm{x} & \mathrm{y} & \mathrm{z} & \mathrm{z/d} \end{array}\right].$$

Для перехода к декартовым координатам делим все на z/d и получаем:

$$[\ \mathrm{X/(Z/d)}\ \ \mathrm{Y/(Z/d)}\ \ \mathrm{d}\ \ 1\ ].$$

Если же точка просмотра расположена в плоскости проекции, тогда центр проекции расположен в точке (0, 0, $-d$). Рассматривая подобные треугольники, аналогично вышеописанному, можем получить:

$$\mathrm{X}_1 = \frac{\mathrm{X}}{\mathrm{Z/d} + 1}; \qquad \mathrm{Y1} = \frac{\mathrm{Y}}{\mathrm{Z/d} + 1}.$$

Матрица преобразования в этом случае имеет вид:

$$M_0 = \left[\begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1/\mathrm{d} \\ 0 & 0 & 0 & 1 \end{array}\right].$$

Матрица M_0 может быть представлена в виде:

$$\mathrm{M}_0 \ = \ \mathrm{T}(0,0,\mathrm{d}) \cdot \mathrm{M} \cdot \mathrm{T}(0,0,-\mathrm{d}),$$

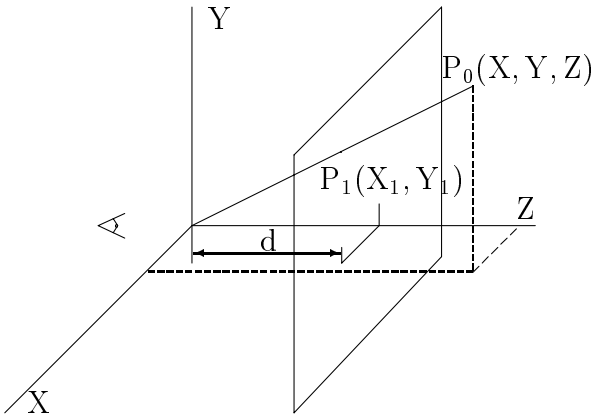


Рис. 0.1.13: Центральная проекция точки P_0 в плоскость $Z = d$

т.е. преобразование проецирования выполняется для этого случая путем переноса начала координат в центр проецирования, собственно проецирования и обратного сдвига начала координат.

0.1.7 Стереоизображения

Существенное повышение наглядности изображения достигается использованием псевдостереоизображений. В этом случае каждым глазом надо рассматривать отдельный перспективный вид. Оба таких вида отображаются на экран дисплея. Для их разделения могут использоваться:

- цветовое разделение, когда, например, изображение для левого глаза строится красным цветом, а для правого — синим и для просмотра используются цветные очки; недостаток этого способа состоит в том, что по сути дела можно формировать только простые каркасные изображения, но зато реализация проста и полностью используется пространственное разрешение дисплея;
- пространственное-временное разделение, когда, например, изображение для левого глаза строится в четных строках, а для правого — в нечетных и для просмотра используются электронные или электромеханические очки, перекрывающие или левый или правый глаз на время прорисовки нечетных, четных строк, соответственно; этот подход может быть реализован на дисплеях с чересстрочной разверткой, позволяет выводить достаточно динамические цветные полутоновые изображения, но с уменьшением вдвое разрешения по вертикали;
- временное разделение, когда для левого глаза используется одна страница видеопамяти, а для правого — вторая и происходит их переключение с достаточно большой частотой кадровой развертки; для перекрытия глаз также должны использоваться электронные или электромеханические очки; пространственное разрешение по строкам здесь не теряется.

Кроме этого, статические цветные псевдостереоизображения могут быть получены последовательным фотографированием изображений для левого и правого глаз с последующим просмотром через стереоскоп.

Используя результаты, полученные в предыдущем пункте, легко сконструировать матрицы преобразований для получения стереопроекций для левого и правого глаз.

0.1.8 Геометрические преобразования растровых картин

Также как и для векторных изображений в двумерном случае, для растровых картин могут

в общем случае требоваться преобразования сдвига, масштабирования и поворота. В связи с существенно дискретным характером изображения при выполнении этих преобразований имеется ряд особенностей.

Преобразование сдвига реализуется наиболее просто и заключается в переписывании части изображения (bitblt — операции Bit Block Transfer). При этом возможно исполнение некоторых операций над старым и новым пикселами с одинаковыми координатами.

Наиболее употребимыми являются:

- замена — новый пиксел просто заменяет старый,
- исключающее ИЛИ — в видеопамять заносится результат операции XOR над старым и новым кодами пикселов. Эта операция обычно используется дважды — вначале для занесения некоторого изображения, например, перекрестия и повторного его занесения для восстановления исходной картины.

Кроме этого, для реализации техники "акварель", т.е. техники работы с прозрачными цветами, в видеопамять заносится результат цветовой интерполяции между старым и новым оттенками пикселов. Эта операция всегда точно реализуема в полноцветных дисплеях, хранящих значения R, G и B в каждом пикселе. В дисплеях с таблицей цветности возможно получение не совсем правильных результатов.

Преобразование масштабирования

Принято различать два типа масштабирования:

- целочисленное — zoom,
- произвольное, когда коэффициент масштабирования не обязательно целое число, — transfocation.

Наиболее просто реализуется целочисленное масштабирование. При увеличении в K раз каждый пиксел в строке дублируется K раз и полученная строка дублируется K раз. При уменьшении в K раз из каждой группы в K строк выбирается одна строка и в ней из каждой группы в K пикселов берется один пиксел в качестве результата. Не целочисленное масштабирование требует нерегулярного дублирования при увеличении и выбрасывания при уменьшении. Для отсутствия "дырок" в результирующем изображении при любых преобразованиях растровых картин следует для очередного пиксела результирующего изображения определить соответствующий (соответствующие) пикселы исходного изображения, вычислить значение пиксела и занести его.

Рассмотрим нецелочисленное уменьшение, т.е. перепись из большего массива в меньший (увеличение строится похожим образом).

Алгоритм ясен из следующей программы:

```
#define Max 13  // размер исходного массива
#define Min 7   // размер результирующего массива

int   ist;      // целая часть приращения индекса большего
                // массива при смещении на 1 по меньшему
float rst;      // дробная часть приращения индекса большего
                // массива при смещении на 1 по меньшему
int   iwr;      // индекс записи
int   ird;      // целая часть индекса чтения
float rrd;      // дробная часть индекса чтения
```

```
char  isx[Max]; // исходный массив пикселов
char  rez[Min]; // результирующий массив пикселов

void main (void) {
   ist= (int)(Max / Min);
   rst= (float)(Max - ist*Min)/(float)Min;
   ird= 0;  rrd= 0.0;
   for (iwr=0; iwr < Min; iwr++) {
      ird= ird + ist;   // накопление целой части индекса;
      rrd= rrd + rst;   // накопление дробной части индекса;
      if (rrd >= 1.0) {rrd= rrd - 1.0;  ird= ird + 1; }
      rez[iwr]= rez[ird-1];
   }
}
```

Понятно, что такой алгоритм требует точной вещественной арифметики, версия алгоритма с целочисленной арифметикой имеет вид:

```
#define Max 13  // размер исходного массива
#define Min 7   // размер результирующего массива

int   ist;      // целая часть приращения индекса большего
                // массива при смещении на 1 по меньшему
int   rst;      // остаток от приращения индекса
                // меньшего массива
int   iwr;      // индекс записи
int   ird;      // целая часть индекса чтения
int   rrd;      // остаток индекса чтения
char  isx[Max]; // исходный массив пикселов
char  rez[Min]; // результирующий массив пикселов

void main (void) {
   ist= (int)(Max / Min);
   rst= Max - ist*Min;
   ird= 0;  rrd= 0.0;
   for (iwr=0; iwr < Min; iwr++) {
      ird= ird + ist;   // накопление целой части индекса;
      rrd= rrd + rst;   // накопление дробной части индекса;
      if (rrd >= Min) {rrd= rrd - Min;  ird= ird + 1; }
      rez[iwr]= rez[ird-1];
   }
}
```

Внутренняя часть цикла при записи на ассемблере существенно упрощается, если использовать то, что для обычных 16-ти разрядных ЭВМ при переполнении происходит смена знака.

Преобразование поворота

Определенные проблемы, связанные с дискретных характером изображения, возникают и при повороте растровой картины на угол не кратный 90°. Здесь возможны два подхода:

1. Сканируются строки исходной картины при этом вычисляются новые значения координат пикселов для результирующей картины. Ясно что отсутствие дырок на результирующем изображении может быть обеспечено только при использовании вещественной арифметики, кроме этого возможно повторное занесение пикселов.

2. Сканируются строки результирующей картины и по координатам очередного пиксела определяются координаты пиксела из исходного изображения. Этот подход гарантирует отсутствие дырок, кроме того исключает повторное занесение пикселов.

0.2 ГЕНЕРАЦИЯ ВЕКТОРОВ

Назначение генератора векторов — соединение двух точек изображения отрезком прямой.

Далее будут рассмотрены четыре алгоритма:

- два алгоритма ЦДА — цифрового дифференциального анализатора (DDA — Digital Differential Analyzer) для генерации векторов — обычный и несимметричный;
- алгоритм Брезенхема для генерации векторов[25];
- алгоритм Брезенхема для генерации ребер заполненного многоугольника с уменьшением ступенчатости.

Перед рассмотрением конкретных алгоритмов сформулируем общие требования к изображению отрезка:

- концы отрезка должны находиться в заданных точках;
- отрезки должны выглядеть прямыми,
- яркость вдоль отрезка должна быть постоянной и не зависеть от длины и наклона.

Ни одно из этих условий не может быть точно выполнено на растровом дисплее в силу того, что изображение строится из пикселов конечных размеров, а именно:

- концы отрезка в общем случае располагаются на пикселах, лишь наиболее близких к требуемым позициям и только в частных случаях координаты концов отрезка точно совпадают с координатами пикселов;
- отрезок аппроксимируется набором пикселов и лишь в частных случаях вертикальных, горизонтальных и отрезков под 45° они будут выглядеть прямыми, причем гладкими прямыми, без ступенек только для вертикальных и горизонтальных отрезков (рис. 0.2.1);
- яркость для различных отрезков и даже вдоль отрезка в общем случае различна, так как, например, расстояние между центрами пикселов для вертикального отрезка и отрезка под 45° различно (см. рис. 0.2.1).

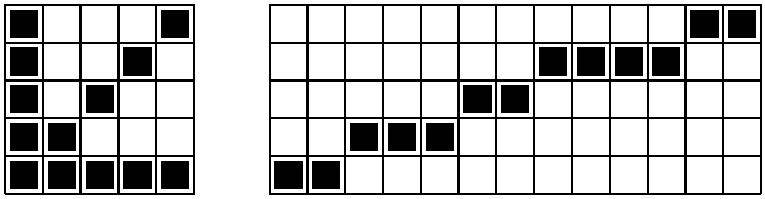


Рис. 0.2.1: Растровое представление различных векторов

Объективное улучшение аппроксимации достигается увеличением разрешения дисплея, но в силу существенных технологических проблем разрешение для растровых систем приемлемой скорости разрешение составляет порядка 1280×1024.

Субъективное улучшение аппроксимации основано на психофизиологических особенностях зрения и, в частности, может достигаться просто уменьшением размеров экрана. Другие способы субъективного улучшения качества аппроксимации основаны на различных программных ухищрениях по “размыванию” резких границ изображения.

Далее в этом разделе рассмотрены три алгоритма генерации отрезка.

0.2.1 Цифровой дифференциальный анализатор

С помощью ЦДА решается дифференциальное уравнение отрезка, имеющее вид:

$$\frac{dY}{dX} = \frac{Py}{Px},$$

где Py = Yk - Yn — приращение координат отрезка по оси Y, а Px = Xk - Xn — приращение координат отрезка по оси X.

При этом ЦДА формирует дискретную аппроксимацию непрерывного решения этого дифференциального уравнения.

В обычном ЦДА, используемом, как правило, в векторных устройствах, тем или иным образом определяется количество узлов N, используемых для аппроксимации отрезка. Затем за N циклов вычисляются координаты очередных узлов:

$$X_0 = Xn; \;\; X_{i+1} = X_i + Px/N.$$

$$Y_0 = Yn; \;\; Y_{i+1} = Y_i + Py/N.$$

Получаемые значения X_i, Y_i преобразуются в целочисленные значения координаты очередного подсвечиваемого пиксела либо округлением, либо отбрасыванием дробной части.

Генератор векторов, использующий этот алгоритм, имеет тот недостаток, что точки могут прописываться дважды, что увеличивает время построения.

Кроме того из-за независимого вычисления обеих координат нет предпочтительных направлений и построенные отрезки кажутся не очень красивыми.

Аппаратная реализация этого алгоритма изложена в пункте 8.1 первой части курса.

Субъективно лучше смотрятся вектора с единичным шагом по большей относительной координате (несимметричный ЦДА). Для Px > Py (при Px, Py > 0) это означает, что координата по X направлению должна увеличиться на 1 Px раз, а координата по Y-направлению должна также Px раз увеличиться, но на Py/Px.

Т.е. количество узлов аппроксимации берется равным числу пикселов вдоль наибольшего приращения.

Для генерации отрезка из точки (x1,y1) в точку (x2,y2) в первом октанте (Px ≥ Py ≥ 0) алгоритм несимметричного ЦДА имеет вид:

1. Вычислить приращения координат:
   Px= x2 - x1;
   Py= y2 - y1;
2. Занести начальную точку отрезка
   PutPixel (x1, y1);
3. Сгенерировать отрезок
   while (x1 < x2) {
   x1:= x1 + 1.0;
   y1:= y1 + Py/Px;
   PutPixel (x1, y1);
   }

Пример генерации отрезка по алгоритму несимметричного ЦДА приведен на рис. 0.2.2.

В Приложении 2 приведена программа V_DDA, реализующая данный алгоритм.

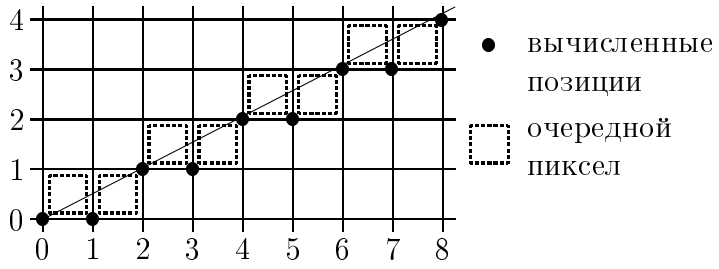


Рис. 0.2.2: Генерация отрезка несимметричным ЦДА

0.2.2 Алгоритм Брезенхема

Так как приращения координат, как правило, не являются целой степенью двойки, то в ЦДА-алгоритме (см. предыдущий пункт) требуется выполнение деления, что не всегда желательно, особенно при аппаратной реализации.

Брезенхем в работе [25] предложил алгоритм, не требующий деления, как и в алгоритме несимметричного ЦДА, но обеспечивающий минимизацию отклонения сгенерированного образа от истинного отрезка, как в алгоритме обычного ЦДА. Основная идея алгоритма состоит в том, что если угловой коэффициент прямой $< 1/2$, то естественно точку, следующую за точкой (0,0), поставить в позицию (1,0) (рис. 0.2.3а), а если угловой коэффициент $> 1/2$, то — в позицию (1,1) (рис. 0.2.3б). Для принятия решения куда заносить очередной пиксел вводится величина отклонения E точной позиции от середины между двумя возможными растровыми точками в направлении наименьшей относительной координаты. Знак E используется как критерий для выбора ближайшей растровой точки.

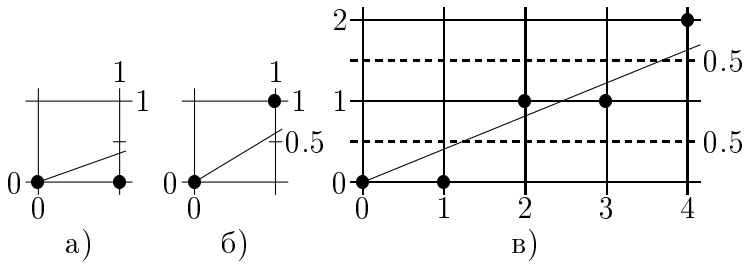


Рис. 0.2.3: Алгоритм Брезенхема генерации отрезков

Если E < 0, то точное Y-значение округляется до последнего меньшего целочисленного значения Y, т.е. Y-координата не меняется по сравнению с предыдущей точкой. В противном случае Y увеличивается на 1.

Для вычисления E без ограничения общности упрощающе положим, что рассматриваемый вектор начинается в точке (0,0) и проходит через точку (4, 1.5) (см. рис. 0.2.3в), т.е. имеет положительный наклон меньший 1.

Из рис. 0.2.3в видно, отклонение для первого шага:

$$_1 = \mathrm{Py}/\mathrm{Px} - 1/2 < 0,$$

поэтому для занесения пиксела выбирается точка (1,0).

Отклонение для второго шага вычисляется добавлением приращения Y-координаты для следующей X-позиции (см. рис. 0.2.3в):

$$_2 = _1 + \mathrm{Py}/\mathrm{Px} > 0,$$

поэтому для занесения пиксела выбирается точка (2,1). Так как отклонение считается от Y-координаты, которая теперь увеличилась на 1, то из накопленного отклонения для вычисления последующих отклонений надо вычесть 1:

$$_2 = _2 - 1.$$

Отклонение для третьего шага:

$$_3 = _2 + \mathrm{Py}/\mathrm{Px} < 0,$$

поэтому для занесения пиксела выбирается точка (3,1).

Суммируя и обозначая большими буквами растровые точки, а маленькими — точки вектора, получаем:

$$_1 = \mathrm{y}_1 - 1/2 = \mathrm{dY}/\mathrm{dX} - 1/2.$$

Возможны случаи:

$_1 > 0$	$\mathrm{E}_1 \le 0$
ближайшая точка есть:	
$\mathrm{X}_1 = \mathrm{X}_0 + 1;\ \ \mathrm{Y}_1 = \mathrm{Y}_0 + 1;$	$\mathrm{X}_1 = \mathrm{X}_0 + 1;\ \ \mathrm{Y}_1 = \mathrm{Y}_0;$
$\mathrm{E}_2 = _1 + \mathrm{Py}/\mathrm{Px} - 1;$	$\mathrm{E}_2 = \mathrm{E}_1 + \mathrm{Py}/\mathrm{Px}.$

Так как интересует только знак E, то можно избавиться от неудобные частных умножением E на 2×Px:

$$\mathrm{E}_1 = 2 \times \mathrm{Py} - \mathrm{Px}$$
$$\mathrm{E}_1 > 0: \quad \mathrm{E}_2 = \mathrm{E}_1 + 2 \times (\mathrm{Py} - \mathrm{Px})$$
$$\mathrm{E}_1 \le 0: \quad \mathrm{E}_2 = \mathrm{E}_1 + 2 \times \mathrm{Py}$$

Таким образом получается алгоритм, в котором используются только целые числа, сложение, вычитание и сдвиг:

```
X= x1;
Y= y1;
Px= x2 − x1;
Py= y2 − y1;
E= 2*Py − Px;
i= Px;
PutPixel(X, Y);        /* Первая точка вектора */
while (i= i−1 ≥ 0) {
    if (E ≥ 0) {
        X= X + 1;
        Y= Y + 1;
        E= E + 2*(Py − Px);
    } else
        X= X + 1;
```

```
        E= E + 2*Py;
    PutPixel(X, Y);      /* Очередная точка вектора */
}
```

Этот алгоритм пригоден для случая $0 \leq dY \leq dX$. Для других случаев алгоритм строится аналогичным образом.

На рис. 0.2.4 приведен пример генерации по алгоритму Брезенхема того же самого отрезка, что и показанного на рис. 0.2.2 для генерации по алгоритму несимметричного ЦДА. Из сравнения рисунков видно, что результаты различны.

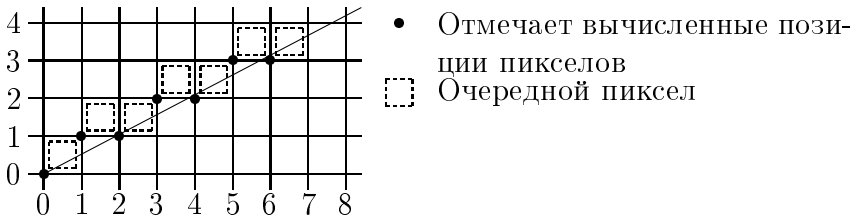


Рис. 0.2.4: Генерация отрезка по алгоритму Брезенхема

В Приложении 2 приведена программа V_BRE, реализующая описанный выше алгоритм.

Разработаны алгоритмы цифрового генератора для окружностей и других конических сечений.

0.2.3 Улучшение качества аппроксимации векторов

Выше было отмечено, что растровая генерация отрезков имеет следующие недостатки:
- неточное расположение начала и конца,
- ступенчатый вид отрезка,
- яркость зависит от наклона и длины.

Ясно, что первый недостаток не может быть устранен программным путем. Неравномерность яркости обычно не слишком заметна и может быть скомпенсирована очевидным образом, требующим, в общем случае, вещественной арифметики, но в связи с реально небольшим количеством различимых уровней яркости достаточно обойтись целочисленным приближением, причем очень грубым.

Наиболее заметно ухудшает качество изображения ступенчатость. Имеется следующие способы борьбы со ступенчатостью :
- увеличение пространственного разрешения за счет усовершенствования аппаратуры,
- трактовка пиксела не как точки, а как площадки конечного размера, яркость которой зависит от размера площади пиксела, занятой изображением отрезка,
- “размывание” резкой границы, за счет частичной подсветки пикселов, примыкающих к формируемому отрезку. Понятно, что при этом ухудшается пространственное разрешение изображения.

В данном пункте мы рассмотрим модифицированный алгоритм Брезенхема, трактующий пиксел как конечную площадку. В следующем пункте будет рассмотрен общий подход, использующий низкочастотную фильтрацию для “размывания” границ изображения.

Модифицированный алгоритм Брезенхема

Основная идея алгоритма состоит в том, чтобы для ребер многоугольника устанавливать яркость пиксела пропорционально площади пиксела, попавшей внутрь многоугольника.

На рис. 0.2.5 приведена иллюстрация построения ребра многоугольника с тангенсом угла наклона 11/21.

На рис. 0.2.5а) показан результат генерации многоугольника с использованием ранее рассмотренного алгоритма Брезенхема при двухуровневом изображении (пиксел или закрашен или не закрашен).

На рис. 0.2.5б) показан результат генерации многоугольника с вычислением интесивности пикселов, через которые проходит ребро многоугольника. Интенсивность вычисляется пропорциональной площади части пиксела, попавшей внутрь многоугольника.

На рис. 0.2.5в) показан результат генерации многоугольника с вычислением интесивности пиксела, через который проходит ребро многоугольника в соответствии с модифицированным алгоритмом Брезенхема.

Как видно из рис. 0.2.5 при построении ребра многоугольника с тангенсом угла наклона t ($0 \leq t \leq 1$) могут захватываться либо один пиксел (пикселы (0,0), (2,1), (4,2), (6,8)) либо два пиксела (пикселы (1,0) и (1,1), (3,1) и (3,2), (5,2) и (5,3)). Если захватывается один пиксел, то часть его площади, попавшая внутрь многоугольника, равна $dy + t/2$ (рис. 0.2.6а).

Если же захватывается два пиксела, то часть площади нижнего пиксела, попавшая внутрь многоугольника равна $1 - \frac{(1-dy)^2}{2t}$, а верхнего — $\frac{(dy-1+2)^2}{2t}$ (см. рис. 0.2.6б). Суммарная площадь частей для двух пикселов, попавшая внутрь многоугольника, равна $dy + t/2$.

Если теперь в исходном алгоритме Брезенхема (см. 0.2.2) заменить отклонение E на $E' = E + (1 - t)$, то $0 \leq E' \leq 1$) и значение E' будет давать значение той части площади пиксела, которая находится внутри многоугольника.

Выполняя преобразование над значением отклонения для первого шага (см. 0.2.2) получим, что начальное значение станет равным 1/2. Максимальное значение отклонения E'_{max}, при превышении которого производится увеличение Y-координаты занесения пикселов, станет равным $(1 - t)$.

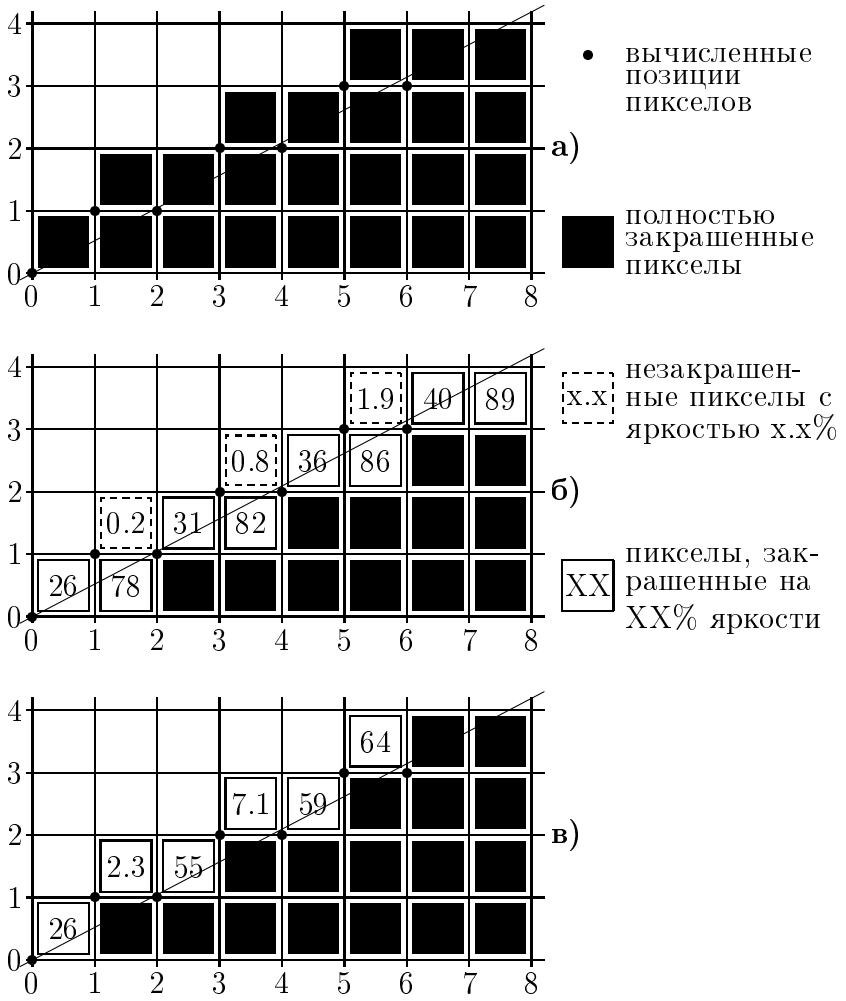


Рис. 0.2.5: Устранение ступенчатости ребер многоугольника: а) генерация ребер без устранения ступенчатости; б) точное вычисление интенсивности пикселов границы; в) формирование пикселов границы по модифицированному методу Брезенхема.

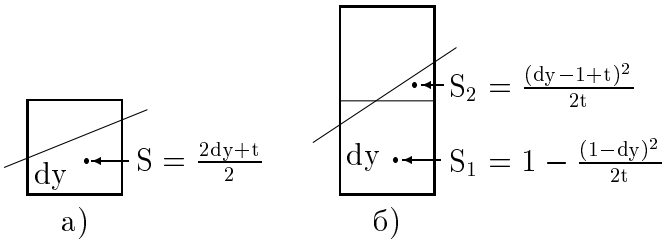


Рис. 0.2.6: Устранение ступенчатости за счет учета площади пикселов, пересекаемых ребром многоугольника.

Для того, чтобы оперировать не дробной частью максимальной интенсивности, а непосредственно ее значениями достаточно домножить на полное число уровней интенсивности I тангенс угла наклона (t), начальное (E') и максимальное (E'_{max}) значения отклонения.

В результате получается следующий алгоритм, пригодный для случая $0 \leq dY \leq dX$:

```
X= x1;
Y= y1;
Px= x2 − x1;
Py= y2 − y1;
t= I ∗ Py/Px;
E′= I/2;
E′max= I − I ∗ Py/Px;
i= Px;
PutPixel(X, Y, t/2);        /* Первая точка вектора */
while (i = i − 1 ≥ 0) {
    if (E′ ≥ E′max) {
        X= X + 1;
        Y= Y + 1;
        E′ = E′ − E′max;
    } else
        X= X + 1;
        E′ = E′ + t;
    PutPixel(X, Y, E′);      /* Очередная точка вектора */
}
```

Для избавления от вещественной арифметики при манипулировании с E' можно домножить уже упомянутые величины на $2 * Px$. Но в этом случае при занесении пикселов потребуется деление E' на $2 * Px$.

В Приложении 2 приведена процедура V_BreM, реализующая модифицированный алгоритм Брезенхема для генерации ребра заполненного многоугольника и пригодная для любого октанта.

0.2.4 Улучшение качества изображения фильтрацией

Мы здесь рассмотрим методы, основанные на “размывании” границы.

Один из них заключается в том, что изображение строится с большим пространственным разрешением, чем позволяет дисплей. При выводе на экран атрибуты пиксела экрана вычи-

сляются усреднением по группе пикселов изображения, построенного с большим разрешением. Т.е. пикселы изображения рассматриваются как подпикселы соответствующих пикселов экрана. Усредняющая маска перемещается по изображению с шагами, равными ее размеру.

Другой метод заключается в усреднении изображения без изменения его разрешения. В этом случае усредняющая маска перемещается по изображению с единичными шагами.

Очевидно, что первый метод должен давать более качественное изображение, но при больших затратах ресурсов. Для усреднения предложены различные маски ([Род89, Прэ82]).

Простейшее усреднение — равномерное (рис. 0.2.7). Улучшить усреднение можно за счет использования весов, задающих влияние отдельных подпикселов на атрибут пиксела экрана (см. рис. 0.2.8).

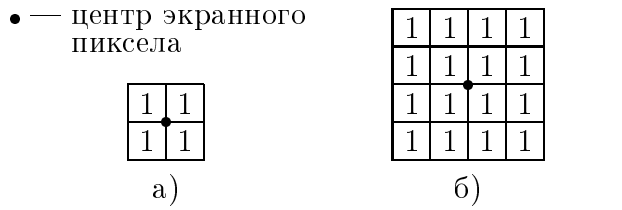


Рис. 0.2.7: Маски для равномерного усреднения изображения

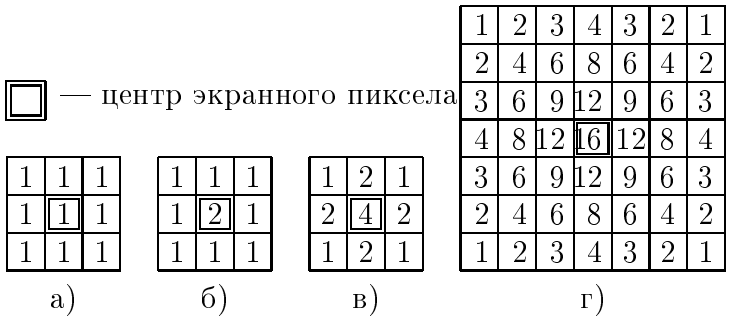


Рис. 0.2.8: Маски для взвешенного усреднения изображения

Эти массивы должны быть пронормированы для получения единичного коэффициента передачи, чтобы не вызывать неправильного смещения средней яркости изображения. Нормирующий коэффициент равен 1 / (сумму членов массива).

Ясно, что эти преобразования, примененные изображению зашумленному случайными импульсными помехами, будут подавлять шум. Это так называемые низкочастотные фильтры.

В Приложении 3 приведено несколько процедур фильтрации, реализующих оба рассмотренных метода — с понижением и без понижения разрешения.

0.3 ГЕНЕРАЦИЯ ОКРУЖНОСТИ

Во многих областях приложений, таких как, например, системы автоматизированного проектирования машиностроительного направления, естественными графическими примитивами, кроме отрезков прямых и строк текстов, являются и конические сечения, т.е. окружности, эллипсы, параболы и гиперболы. Наиболее употребительным примитивом, естественно, является окружность. Один из наиболее простых и эффективных алгоритмов генерации окружности разработан Брезенхемом [26]. В переводной литературе он изложен, в частности, в [Род89, Фол85].

0.3.1 Алгоритм Брезенхема

Для простоты и без ограничения общности рассмотрим генерацию 1/8 окружности, центр которой лежит в начале координат. Остальные части окружности могут быть получены последовательными отражениями (использованием симметрии точек на окружности относительно центра и осей координат).

Окружность с центром в начале координат описывается уравнением:

$$X^2 + Y^2 = R^2$$

Алгоритм Брезенхема пошагово генерирует очередные точки окружности, выбирая на каждом шаге для занесения пиксела точку растра $P_i(X_i,\ Y_i)$, ближайшую к истинной окружности, так чтобы ошибка:

$$E_i(P_i) = (X_i^2 + Y_i^2) - R^2$$

была минимальной. Причем, как и в алгоритме Брезенхема для генерации отрезков, выбор ближайшей точки выполняется с помощью анализа значений управляющих переменных, для вычисления которых не требуется вещественной арифметики. Для выбора очередной точки достаточно проанализировать знаки.

Рассмотрим генерацию 1/8 окружности по часовой стрелке, начиная от точки X=0, Y=R.

Проанализируем возможные варианты занесения i+1-й точки, после занесения i-й.

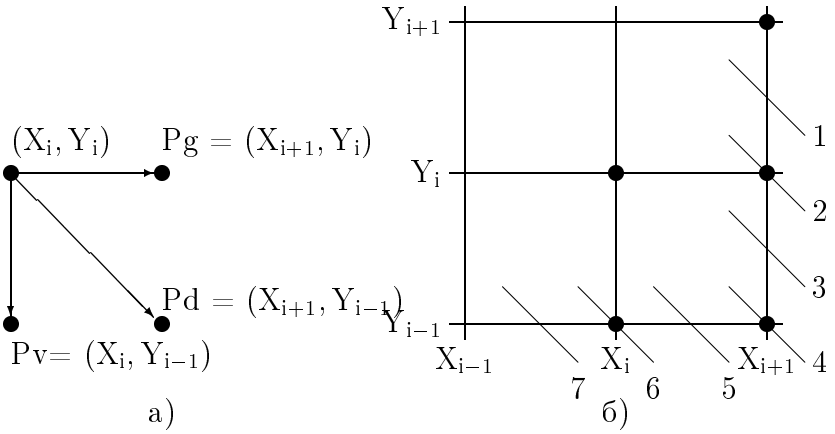


Рис. 0.3.1: Варианты расположения очередного пиксела окружности

При генерации окружности по часовой стрелке после занесения точки (X_i, Y_i) следующая точка может быть (см. рис. 0.3.1а) либо $Pg = (X_{i+1}, Y_i)$ — перемещение по горизонтали, либо $Pd = (X_{i+1}, Y_{i-1})$ — перемещение по диагонали, либо $Pv = (X_i, Y_{i-1})$ — перемещение по вертикали.

Для этих возможных точек вычислим и сравним абсолютные значения разностей квадратов расстояний от центра окружности до точки и окружности:

$$\begin{array}{lcl} |\mathrm{Dg}| & = & |(\mathrm{X}+1)^2 + \mathrm{Y}^2 - \mathrm{R}^2| \\ |\mathrm{Dd}| & = & |(\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2| \\ |\mathrm{Dv}| & = & |\mathrm{X}^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2| \end{array}$$

Выбирается и заносится та точка, для которой это значение минимально.

Выбор способа расчета определяется по значению Dd. Если $\mathrm{Dd} < 0$, то диагональная точка внутри окружности. Это варианты 1–3 (см. рис. 0.3.1б). Если $\mathrm{Dd} > 0$, то диагональная точка вне окружности. Это варианты 5-7. И, наконец, если $\mathrm{Dd} = 0$, то диагональная точка лежит точно на окружности. Это вариант 4. Рассмотрим случаи различных значений Dd в только что приведенной последовательности.

Случай Dd < 0

Здесь в качестве следующего пиксела могут быть выбраны или горизонтальный — Pg или диагональный — Pd.

Для определения того, какой пиксел выбрать Pg или Pd составим разность:

$$\begin{array}{rcl} \mathrm{di} & = & |\mathrm{Dg}| - |\mathrm{Dd}| = \\ & & |(\mathrm{X}+1)^2 + \mathrm{Y}^2 - \mathrm{R}^2| - |(\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2| \end{array}$$

И будем выбирать точку Pg при $\mathrm{di} \le 0$, в противном случае выберем Pd.

Рассмотрим вычисление di для разных вариантов.

Для вариантов 2 и 3:

$\mathrm{Dg} \ge 0$ и $\mathrm{Dd} < 0$, так как горизонтальный пиксел либо вне, либо на окружности, а диагональный внутри.

$$\mathrm{di} = (\mathrm{X}+1)^2 + \mathrm{Y}^2 - \mathrm{R}^2 + (\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2;$$

Добавив и вычтя $(\mathrm{Y}-1)^2$ получим:

$$\mathrm{di} = 2 \cdot \left[(\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2\right] + 2 \cdot \mathrm{Y} - 1$$

В квадратных скобках стоит Dd, так что

$$\mathrm{di} = 2 \cdot (\mathrm{Dd} + \mathrm{Y}) - 1$$

Для варианта 1:

Ясно, что должен быть выбран горизонтальный пиксел Pg. Проверка компонент di показывает, что $\mathrm{Dg} < 0$ и $\mathrm{Dd} < 0$, причем $\mathrm{di} < 0$, так как диагональная точка больше удалена от окружности, т.е. по критерию $\mathrm{di} < 0$ как и в предыдущих случаях следует выбрать горизонтальный пиксел Pg, что верно.

Случай Dd > 0

Здесь в качестве следующего пиксела могут быть выбраны или диагональный — Pd или вертикальный Pv.

Для определения того, какую пиксел выбрать Pd или Pv составим разность:

$$\begin{aligned} \mathrm{si} &= |\mathrm{Dd}| - |\mathrm{Dv}| = \\ &\quad |(\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2| - |\mathrm{X}^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2| \end{aligned}$$

Если $\mathrm{si} \le 0$, то расстояние до вертикальной точки больше и надо выбирать диагональный пиксел Pd, если же $\mathrm{si} > 0$, то выбираем вертикальный пиксел Pv.

Рассмотрим вычисление si для разных вариантов.

Для вариантов 5 и 6:

$\mathrm{Dd} > 0$ и $\mathrm{Dv} \le 0$, так как диагональный пиксел вне, а вертикальный либо вне либо на окружности.

$$\mathrm{si} = (\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2 + \mathrm{X}^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2;$$

Добавив и вычтя $(\mathrm{X}+1)^2$ получим:

$$\mathrm{si} = 2 \cdot \left[(\mathrm{X}+1)^2 + (\mathrm{Y}-1)^2 - \mathrm{R}^2\right] - 2 \cdot \mathrm{X} - 1$$

В квадратных скобках стоит Dd, так что

$$\mathrm{si} = 2 \cdot (\mathrm{Dd} - \mathrm{X}) - 1$$

Для варианта 7:

Ясно, что должен быть выбран вертикальный пиксел Pv. Проверка компонент si показывает, что $\mathrm{Dd} > 0$ и $\mathrm{Dv} > 0$, причем $\mathrm{si} > 0$, так как диагональная точка больше удалена от окружности, т.е. по критерию $\mathrm{si} > 0$ как и в предыдущих случаях следует выбрать вертикальный пиксел Pv, что соответствует выбору для вариантов 5 и 6.

Случай Dd = 0

Для компонент di имеем: $\mathrm{Dg} > 0$ и $\mathrm{Dd} = 0$, следовательно по критерию $\mathrm{di} > 0$ выбираем диагональный пиксел.

С другой стороны, для компонент si имеем: $\mathrm{Dd} = 0$ и $\mathrm{Dv} < 0$, так что по критерию $\mathrm{si} \le 0$ также выбираем диагональный пиксел.

Итак:

$\mathrm{Dd} < 0$

$\mathrm{di} \le 0$ — выбор горизонтального пиксела Pg

$\mathrm{di} > 0$ — выбор диагонального пиксела Pd

$\mathrm{Dd} > 0$

$\mathrm{si} \le 0$ — выбор диагонального пиксела Pd

$\mathrm{si} > 0$ — выбор вертикального пиксела Pv

$Dd = 0$

выбор диагонального пиксела Pd.

Выведем рекуррентные соотношения для вычисления Dd для (i+1)-го шага, после выполнения i-го.

1. Для горизонтального шага к X_{i+1}, Y_i

$X_{i+1} = X_i + 1$

$Y_{i+1} = Y_i$

$Dd_{i+1} = (X_{i+1} + 1)^2 + (Y_{i+1} - 1)^2 - R^2 =$

$X_{i+1}{}^2 + 2 \cdot X_{i+1} + 1 + (Y_{i+1} - 1)^2 - R^2 =$

$(X_i + 1)^2 + (Y_i - 1)^2 - R^2 + 2 \cdot X_{i+1} + 1 =$

$Dd_i + 2 \cdot X_{i+1} + 1$

2. Для диагонального шага к X_{i+1}, Y_{i-1}

$X_{i+1} = X_i + 1$

$Y_{i+1} = Y_i - 1$

$Dd_{i+1} = Dd_i + 2 \cdot X_{i+1} - 2 \cdot Y_{i+1} + 2$

3. Для вертикального шага к X_i, Y_{i-1}

$X_{i+1} = X_i$

$Y_{i+1} = Y_i - 1$

$Dd_{i+1} = Dd_i - 2 \cdot Y_{i+1} + 1$

В Приложении 5 приведена подпрограмма V_circle, реализующая описанный выше алгоритм и строящая дугу окружности в первой четверти. Начальная инициализация должна быть:

$X = 0$

$Y = R$

$Dd = (X + 1)^2 + (Y - 1)^2 - R^2 = 1 + (R - 1)^2 - R^2 = 2 * (1 - R)$

Пикселы в остальных четвертях можно получить отражением. Кроме того достаточно сформировать дугу только во втором октанте, а остальные пикселы сформировать из соображений симметрии, например, с помощью подпрограммы Pixel_circle, приведенной в Приложении 5 и заносящей симметричные пикселы по часовой стрелке от исходного.

В Приложении 6 приведены подпрограмма V_BRcirc, реализующая описанный выше алгоритм и строящая дугу окружности во втором октанте с последующим симметричным занесением пикселов. Эта процедура может строить и 1/4 окружности. Подробнее см. текст Приложения 6. Там же приведена более короткая подпрограмма, строящая 1/8 окружности методом Мичнера [4], (том 1, стр. 152). Остальная часть окружности строится симметрично.

0.4 ЗАПОЛНЕНИЕ МНОГОУГОЛЬНИКА

В большинстве приложений используется одно из существенных достоинств растровых устройств — возможность заполнения областей экрана.

Существует две разновидности заполнения:

- первая, связанная как с интерактивной работой, так и с программным синтезом изображения, служит для заполнения внутренней части многоугольника, заданного координатами его вершин.
- вторая, связанная в первую очередь с интерактивной работой, служит для заливки области, которая либо очерчена границей с кодом пиксела, отличающимся от кодов любых пикселов внутри области, либо закрашена пикселами с заданным кодом;

В данном разделе рассмотрим алгоритм заполнения многоугольника. В следующем разделе будут рассмотрены алгоритмы заливки области.

Простейший способ заполнения многоугольника, заданного координатами вершин, заключается в определении принадлежит ли текущий пиксел внутренней части многоугольника. Если принадлежит, то пиксел заносится.

Определить принадлежность пиксела многоугольнику можно, например, подсчетом суммарного угла с вершиной на пикселе при обходе контура многоугольника. Если пиксел внутри, то угол будет равен 360°, если вне — 0° (рис. 0.4.1).

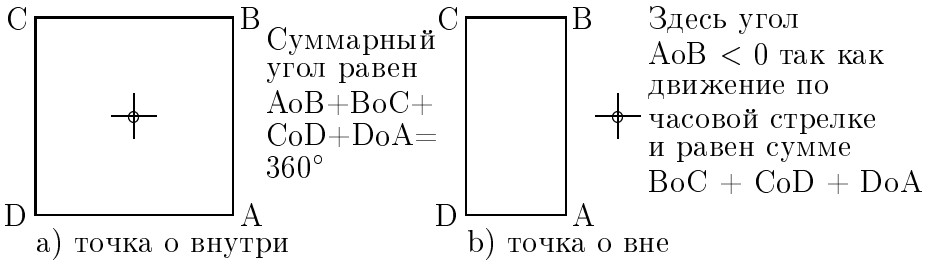


Рис. 0.4.1: Определение принадлежности пиксела многоугольнику

Вычисление принадлежности должно производиться для всех пикселов экрана и так как большинство пикселов скорее всего вне многоугольников, то данный способ слишком расточителен. Объем лишних вычислений в некоторых случаях можно сократить использованием прямоугольной оболочки — минимального прямоугольника, объемлющего интересующий объект, но все равно вычислений будет много. Другой метод определения принадлежности точки внутренней части многоугольника будет рассмотрен ниже при изучении отсечения отрезков по алгоритму Кируса-Бека.

0.4.1 Построчное заполнение

Реально используются алгоритмы построчного заполнения, основанные на том, что соседние пикселы в строке скорее всего одинаковы и меняются только там где строка пересекается с ребром многоугольника. Это называется когерентностью растровых строк (строки сканирования Y_i, Y_{i+1}, Y_{i+2} на рис. 0.4.2). При этом достаточно определить X-координаты пересечений

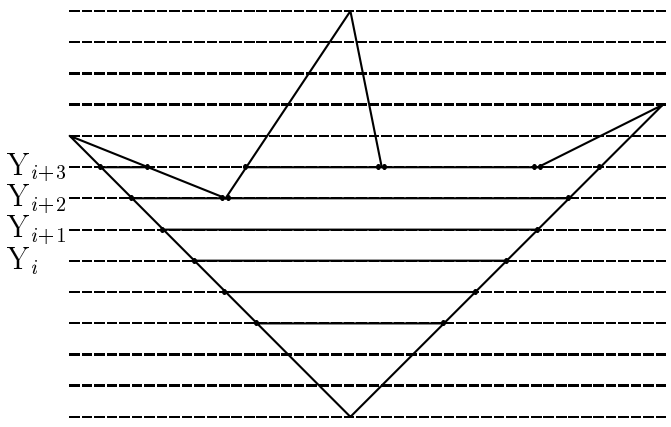


Рис. 0.4.2: Построчная закраска многоугольника

строк сканирования с ребрами. Пары отсортированных точек пересечения задают интервалы заливки.

Кроме того, если какие-либо ребра пересекались i-й строкой, то они скорее всего будут пересекаться также и строкой i+1. (строки сканирования Y_i и Y_{i+1} на рис. 0.4.2). Это называется когерентностью ребер. При переходе к новой строке легко вычислить новую X-координату точки пересечения ребра, используя X-координату старой точки пересечения и тангенс угла наклона ребра:

$$X_{i+1} = X_i + 1/k$$

(тангенс угла наклона ребра — $k = dy/dx$, так как $dy = 1$, то $1/k = dx$).

Смена же количества интервалов заливки происходит только тогда, когда в строке сканирования появляется вершина.

Учет когерентности строк и ребер позволяет построить для заполнения многоугольников различные высокоэффективные алгоритмы построчного сканирования. Для каждой строки сканирования рассматриваются только те ребра, которые пересекают строку. Они задаются списком активных ребер (САР). При переходе к следующей строке для пересекаемых ребер перевычисляются X-координаты пересечений. При появлении в строке сканирования вершин производится перестройка САР. Ребра, которые перестали пересекаться, удаляются из САР, а все новые ребра, пересекаемые строкой заносятся в него.

Общая схема алгоритма, динамически формирующего список активных ребер и заполняющего многоугольник снизу-вверх, следующая:

1. Подготовить служебные целочисленные массивы Y-координат вершин и номеров вершин.

2. Совместно отсортировать Y-координаты по возрастанию и массив номеров вершин для того, чтобы можно было определить исходный номер вершины.

3. Определить пределы заполнения по оси Y — Y_min и Y_max. Стартуя с текущим значением Y_tek=Y_min, исполнять пункты 4–9 до завершения раскраски.

4. Определить число вершин, расположенных на строке Y_tek — текущей строке сканирования.

5. Если вершины есть, то для каждой из вершин дополнить список активных ребер, используя информацию о соседних вершинах.
   Для каждого ребра в список активных ребер заносятся:

   - максимальное значение Y-координаты ребра,
   - приращение X-координаты при увеличении Y на 1,
   - начальное значение X-координаты.

   Если обнаруживаются горизонтальные ребра, то они просто закрашиваются и информация о них в список активных ребер не заносится.
   Если после этого обнаруживается, что список активных ребер пуст, то заполнение закончено.

6. По списку активных ребер определяется Y_след — Y-координата ближайшей вершины. (Вплоть до Y_след можно не заботиться о модификации САР а только менять X-координаты пересечений строки сканирования с активными ребрами).

7. В цикле от Y_tek до Y_след:

   - выбрать из списка активных ребер и отсортировать X-координаты пересечений активных ребер со строкой сканирования;
   - определить интервалы и выполнить закраску;
   - перевычислить координаты пересечений для следующей строки сканирования.

8. Проверить не достигли ли максимальной Y-координаты. Если достигли, то заливка закончена, иначе выполнить пункт 9.

9. Очистить список активных ребер от ребер, закончившихся на строке Y_след и перейти к пункту 4.

В Приложении 5 приведены две подпрограммы заполнения многоугольника — V_FP0 и V_FP1. Первая реализует данный (простейший) алгоритм. Эта программа вполне работоспособна, но генерирует двух и трехкратное занесение части пикселов. Это мало приемлемо для устройств вывода типа матричных или струйных принтеров.

В отличие от V_FP0, в программе V_FP1 используется более сложный алгоритм формирования списка активных ребер, обеспечивающий практически полное отсутствие дублирований (рис. 0.4.3).

0.4.2 Сортировка методом распределяющего подсчета

Понятно, что одна из важнейших работ в алгоритме построчного сканирования — сортировка. В связи с заведомо ограниченной разрешающей способностью растровых дисплеев (не более 2048) иногда целесообразно использовать чрезвычайно эффективный алгоритм сортировки методом распределяющего подсчета.

Для рассмотрения алгоритма предположим, что надо отсортировать числа, заданные в массиве с именем “Исходный_массив”; количество сортируемых чисел задается скаляром “Колво_чисел”; сортируемые числа J удовлетворяют условию:

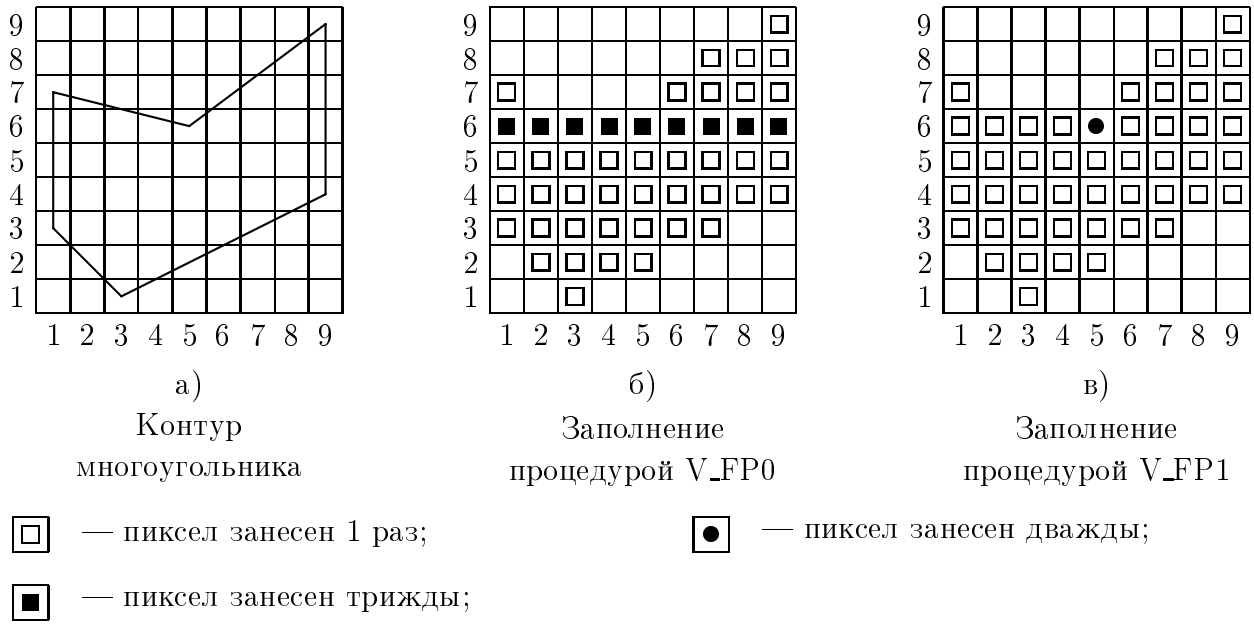


Рис. 0.4.3: Сравнение алгоритмов заполнения многоугольника

$$0 \le \mathrm{J} < \mathrm{Max\_}.$$

Для сортировки потребуются описания:

```
int  Max_число;         /* Верхняя граница значений */
int  *Повтор;           /* Длина этого массива = Max_число */
int  Кол_чисел;         /* Кол-во сортируемых чисел */
int  *Исходный_массив; /* Длина этого массива >= Кол_чисел */
int  *Результат;        /* Длина этого массива >= Кол_чисел */
int  ii,jj, kk;         /* Рабочие переменные */
```

1. Обнуляется служебный массив для подсчета числа повторений исходных кодов.

   ```
   for (ii=0; ii<Max_число; ++ii) Повтор[ii]= 0;
   ```

2. Сортируемый массив просматривается и вычисляется количество раз повторений каждого числа:

   ```
   for (ii= 0; ii < Кол_чисел; ++ii) {
      jj= Исходный_массив[ii];
      Повтор[jj]= Повтор[jj] + 1;
   }
   ```

3. Суммируется количество повторений каждого числа, так что значение Повтор[J] даст начальное расположение группы чисел, равных J, в отсортированном массиве:

```
jj= 0;
for (ii=0; ii<Max_число; ++ii) {
   jj= jj + Повтор[ii];
   Повтор[ii]= jj;
}
```

4. Просматривается исходный массив и числа из него заносятся в массив результатов той же длины. Индекс занесения числа J в массив результатов равен значению J-го элемента массива Повтор. После занесения числа J значение Повтор[J] уменьшается на 1:

```
for (ii= 0; ii < Кол_чисел; ++ii) {
   jj= Исходный_массив[ii];
   kk= Повтор[jj];
   Результат[kk]= jj;
   Повтор[jj]= Повтор[jj] - 1;
}
```

0.5 ЗАЛИВКА ОБЛАСТИ С ЗАТРАВКОЙ

Как уже отмечалось, для приложений, связанных в основном с интерактивной работой, используются алгоритмы заполнения области с затравкой.

При этом тем или иным образом задается заливаемая (перекрашиваемая) область, код пиксела, которым будет выполняться заливка и начальная точка в области, начиная с которой начнется заливка.

По способу задания области делятся на два типа:

- гранично-определенные, задаваемые своей (замкнутой) границей такой, что коды пикселов границы отличны от кодов внутренней, перекрашиваемой части области. На коды пикселы внутренней части области налагаются два условия — они должны быть отличны от кода пикселов границы и кода пиксела перекраски. Если внутри гранично-определенной области имеется еще одна граница, нарисованная пикселами с тем же кодом, что и внешняя граница, то соответствующая часть области не должна перекрашиваться;
- внутренне-определенные, нарисованные одним определенным кодом пиксела. При заливке этот код заменяется на новый код закраски.

В этом состоит основное отличие заливки области с затравкой от заполнения многоугольника. В последнем случае мы сразу имеем всю информацию о предельных размерах части экрана, занятой многоугольником. Поэтому определение принадлежности пиксела многоугольнику базируется на быстро работающих алгоритмах, использующих когерентность строк и ребер (см. предыдущий раздел). В алгоритмах же заливки области с затравкой нам вначале надо прочитать пиксел, затем определить принадлежит ли он области и если принадлежит, то перекрасить.

Заливаемая область или ее граница — некоторое связное множество пикселов. По способам доступа к соседним пикселам области делятся на 4-х и 8-ми связные. В 4-х связных областях доступ к соседним пикселам осуществляется по четырем направлениям — горизонтально влево и вправо и в вертикально вверх и вниз. В 8-ми связных областях к этим направлениям добавляются еще 4 диагональных. Используя связность мы может, двигаясь от точки затравки, достичь и закрасить все пикселы области.

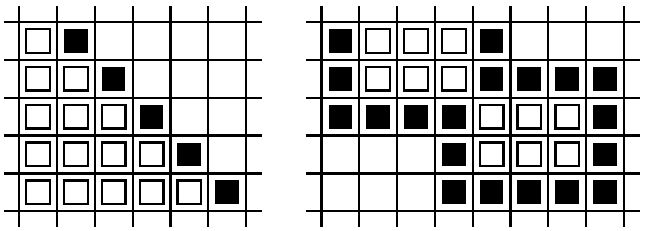
Важно отметить, что для 4-х связной прямоугольной области граница 8-ми связна (рис. 0.5.1а) и наоборот у 8-ми связной области граница 4-х связна (см. рис. 0.5.1б). Поэтому заполнение 4-х связной области 8-ми связным алгоритмом может привести к “просачиванию” через границу и заливке пикселов в примыкающей области.

В общем, 4-х связную область мы можем заполнить как 4-х, так и 8-ми связным алгоритмом. Обратное же неверно. Так область на рис. 0.5.1а мы можем заполнить любым алгоритмом, а область на рис. 0.5.1б, состоящую из двух примыкающих 4-х связных областей можно заполнить только 8-ми связным алгоритмом.

С использованием связности областей и стека можно построить простые алгоритмы закраски как внутренне, так и гранично-определенной области. В [4] рассматриваются совсем короткие рекурсивные подпрограммы заливки. В [3] — несколько более длинные итеративные подпрограммы.

0.5.1 Простой алгоритм заливки

Рассмотрим простой алгоритм заливки гранично-определенной 4-х связной области. В [4] приведена рекурсивная реализация подпрограммы заливки 4-х связной гранично-определенной области:



а) 4-х б) 8-ми
связные, внутренне-определенные области
□ — пикселы области; ■ — пикселы границы.

Рис. 0.5.1: Связность областей и их границ

```
void V_FAB4R (grn_pix, new_pix, x_isx, y_isx)
int grn_pix, new_pix, x_isx, y_isx;
{
      if (getpixel (x_isx, y_isx) ≠ grn_pix &&
      getpixel (x_isx, y_isx) ≠ new_pix)
      {
        putpixel (x_isx, y_isx, new_pix);
        V_FAB4R (grn_pix, new_pix, x_isx+1, y_isx);
        V_FAB4R (grn_pix, new_pix, x_isx, y_isx+1);
        V_FAB4R (grn_pix, new_pix, x_isx-1, y_isx);
        V_FAB4R (grn_pix, new_pix, x_isx, y_isx-1);
      }
} /* V_FAB4R */
```

Заливка выполняется следующим образом:

- определяется является ли пиксел граничным или уже закрашенным,
- если нет, то пиксел перекрашивается, затем проверяются и если надо перекрашиваются 4 соседних пиксела.

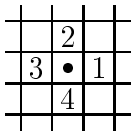
Полный текст тестовой программы V_FAB4R с использованием этой подпрограммы приведен в Приложении 6.

Понятно, что несмотря на простоту и изящество программы, рекурсивная реализация проигрывает итеративной в том, что требуется много памяти для упрятывания вложенных вызовов.

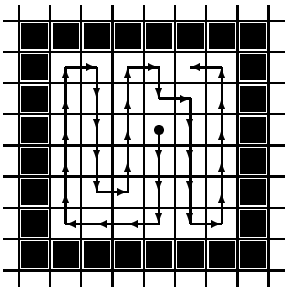
В [3] приведен итеративный алгоритм закраски 4-х связной гранично-определенной области. Логика работы алгоритма следующая:

Поместить координаты затравки в стек
Пока стек не пуст
  Извлечь координаты пиксела из стека.
  Перекрасить пиксел.
  Для всех четырех соседних пикселов проверить
  является ли он граничным или уже перекрашен.
  Если нет, то занести его координаты в стек.

На рис. 0.5.2 а) показан выбранный порядок перебора соседних пикселов, а на рис. 0.5.2 б) соответствующий ему порядок закраски простой гранично-определенной области.



а)
Порядок перебора соседних пикселов



б)
Порядок заливки области

Рис. 0.5.2: Заливка 4-х связной области итеративным алгоритмом

Ясно, что такой алгоритм экономнее, так как в стек надо упрятывать только координаты.

Рассмотренный алгоритм легко модифицировать для работы с 8-ми связными граничноопределенными областями или же для работы с внутренне-определенными.

Программа V_FAB4, реализующая данный алгоритм, приведена в Приложении 6.

Сравнительные прогоны тестовых программ V_FAB4R и V_FAB4 подтвердили соображения о неэкономности рекурсивного алгоритма: при стандартном окне стека в 64 К с помощью рекурсивной программы можно закрасить квадратик не более чем 57×57 пикселов. Итеративная же программа V_FAB4 при тех же условиях позволяет закрасить прямоугольник размером 110×110 истратив на массив координат 16382 байта.

Как уже отмечалось, очевидный недостаток алгоритмов непосредственно использующих связность закрашиваемой области — большие затраты памяти на стек, так как на каждый закрашенный пиксел в стеке по максимуму будет занесена информация о еще трех соседних. Кроме того, информация о некоторых пикселах может записываться в стек многократно. Это приведет не только к перерасходу памяти, но и потере быстродействия за счет многократной раскраски одного и того же пиксела. Значительно более экономен далее рассмотренный построчный алгоритм заливки.

0.5.2 Построчный алгоритм заливки с затравкой

Использует пространственную когерентность:

- пикселы в строке меняются только на границах;
- при перемещении к следующей строке размер заливаемой строки скорее всего или неизменен или меняется на 1 пиксел.

Таким образом, на каждый закрашиваемый фрагмент строки в стеке хранятся координаты только одного начального пиксела [36], что приводит к существенному уменьшению размера стека.

Последовательность работы алгоритма для гранично определенной области следующая:

1. Координата затравки помещается в стек, затем до исчерпания стека выполняются пункты 2-4.

2. Координата очередной затравки извлекается из стека и выполняется максимально возможное закрашивание вправо и влево по строке с затравкой, т.е. пока не попадется граничный пиксел. Пусть это Хлев и Хправ, соответственно.

3. Анализируется строка ниже закрашиваемой в пределах от Хлев до Хправ и в ней находятся крайние правые пикселы всех незакрашенных фрагментов. Их координаты заносятся в стек.

4. То же самое проделывается для строки выше закрашиваемой.

В Приложении 6 приведена процедура V_FAST, реализующая рассмотренный алгоритм. За счет несложной модификации служебных процедур запроса и записи строк изображения, данная процедура может заливать изображение, размещенное в файле.

0.6 ОТСЕЧЕНИЕ ОТРЕЗКОВ

Если изображение выходит за пределы экрана, то на части дисплеев увеличивается время построения за счет того, что изображение строится в "уме". В некоторых дисплеях выход за пределы экрана приводит к искажению картины, так как координаты просто ограничиваются при достижении ими граничных значений, а не выполняется точный расчет координат пересечения (эффект "стягивания" изображения). Некоторые, в основном, простые дисплеи просто не допускают выхода за пределы экрана. Все это, особенно в связи с широким использованием технологии просмотра окнами, требует выполнения отсечения сцены по границам окна видимости.

В простых графических системах достаточно двумерного отсечения, в трехмерных пакетах используется трех и четырехмерное отсечение. Последнее выполняется в ранее рассмотренных однородных координатах, позволяющих единым образом выполнять аффинные и перспективные преобразования.

Программное исполнение отсечения достаточно медленный процесс, поэтому, естественно, в мощные дисплеи встраивается соответствующая аппаратура. Первое сообщение об аппаратуре отсечения, использующей алгоритм отсечения делением отрезка пополам и реализованной в устройстве Clipping Divider, появилось в 1968 г. [38]. Этот алгоритм был рассмотрен при изучении технических средств. Здесь мы рассмотрим программные реализации алгоритма отсечения.

Отсекаемые отрезки могут быть трех классов — целиком видимые, целиком невидимые и пересекающие окно. Очевидно, что целесообразно возможно более рано, без выполнения большого объема вычислений принять решение об видимости целиком или отбрасывании. По способу выбора простого решения об отбрасывании невидимого отрезка целиком или принятия его существует два основных типа алгоритмов отсечения — алгоритмы, использующие кодирование концов отрезка или всего отрезка и алгоритмы, использующие параметрическое представление отсекаемых отрезков и окна отсечения. Представители первого типа алгоритмов — алгоритм Коэна-Сазерленда (Cohen-Sutherland, CS-алгоритм) [4] и FC-алгоритм (Fast Clipping — алгоритм) [37]. Представители алгоритмов второго типа — алгоритм Кируса-Бека (Curus-Beck, CB — алгоритм) и более поздний алгоритм Лианга-Барски (Liang-Barsky, LB-алгоритм) [32].

Алгоритмы с кодированием применимы для прямоугольного окна, стороны которого параллельны осям координат, в то время как алгоритмы с параметрическим представлением применимы для произвольного окна.

Вначале мы рассмотрим алгоритм Коэна-Сазерленда, являющийся стандартом де-факто алгоритма отсечения линий и обладающий одним из лучших быстродействий при компактной реализации. Затем рассмотрим наиболее быстрый, но и чрезвычайно громоздкий FC-алгоритм. Далее рассмотрим алгоритм Лианга-Барски для отсечения прямоугольным окном с использованием параметрического представления. Быстродействие этого алгоритма сравнимо с быстродействием алгоритма Коэна-Сазерленда при большей компактности и наличии 3D и 4D реализаций. Последним рассмотрим алгоритм Кируса-Бека, который использует параметрическое представление и позволяет отсекать произвольным выпуклым окном. В заключение сравним быстродействие различных алгоритмов.

0.6.1 Двумерный алгоритм Коэна-Сазерленда

Этот алгоритм позволяет быстро выявить отрезки, которые могут быть или приняты или отброшены целиком. Вычисление пересечений требуется когда отрезок не попадает ни в один из этих классов. Этот алгоритм особенно эффективен в двух крайних случаях:

- большинство примитивов содержится целиком в большом окне,
- большинство примитивов лежит целиком вне относительно маленького окна.

Идея алгоритма состоит в следующем:

Окно отсечения и прилегающие к нему части плоскости вместе образуют 9 областей (рис. 0.6.1). Каждой из областей присвоен 4-х разрядный код.

Две конечные точки отрезка получают 4-х разрядные коды, соответствующие областям, в которые они попали. Смысл разрядов кода:

1 pp = 1 — точка над верхним краем окна;

2 pp = 1 — точка под нижним краем окна;

3 pp = 1 — точка справа от правого края окна;

4 pp = 1 — точка слева от левого края окна.

Определение того лежит ли отрезок целиком внутри окна или целиком вне окна выполняется следующим образом:

- если коды обоих концов отрезка равны 0 то отрезок целиком внутри окна, отсечение не нужно, отрезок принимается как тривиально видимый (отрезок AB на рис. 0.6.1);
- если логическое & кодов обоих концов отрезка не равно нулю, то отрезок целиком вне окна, отсечение не нужно, отрезок отбрасывается как тривиально невидимый (отрезок KL на рис. 0.6.1);
- если логическое & кодов обоих концов отрезка равно нулю, то отрезок подозрительный, он может быть частично видимым (отрезки CD, EF, GH) или целиком невидимым (отрезок IJ); для него нужно определить координаты пересечений со сторонами окна и для каждой полученной части определить тривиальную видимость или невидимость. При этом для отрезков CD и IJ потребуется вычисление одного пересечения, для остальных (EF и GH) — двух.

При расчете пересечения используется горизонтальность либо вертикальность сторон окна, что позволяет определить координату X или Y точки пересечения без вычислений.

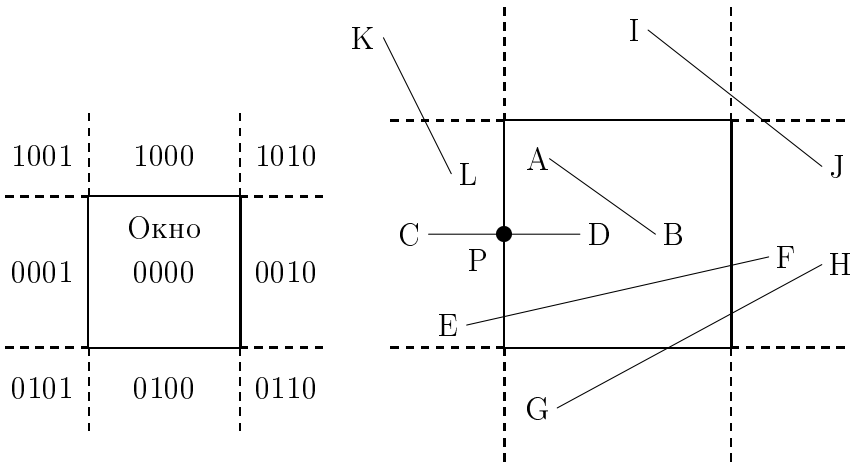


Рис. 0.6.1: Отсечение по методу Коэна-Сазерленда

При непосредственном использовании описанного выше способа отбора целиком видимого или целиком невидимого отрезка после расчета пересечения потребовалось бы вычисление кода расположения точки пересечения. Для примера рассмотрим отрезок CD. Точка пересечения

обозначена как P. В силу того, что граница окна считается принадлежащей окну, то можно просто принять только часть отрезка PD, попавшую в окно. Часть же отрезка CP, на самом деле оказавшаяся вне окна, потребует дальнейшего рассмотрения, так как логическое И кодов точек C и P даст 0, т.е. отрезок CP нельзя просто отбросить. Для решения этой проблемы Коэн и Сазерленд предложили заменять конечную точку с ненулевым кодом конца на точку, лежащую на стороне окна, либо на ее продолжении.

В целом схема алгоритма Коэна-Сазерленда следующая:

1. Рассчитать коды конечных точек отсекаемого отрезка.

   В цикле повторять пункты 2–6:

2. Если логическое И кодов конечных точек не равно 0, то отрезок целиком вне окна. Он отбрасывается и отсечение закончено.

3. Если оба кода равны 0, то отрезок целиком видим. Он принимается и отсечение закончено.

4. Если начальная точка внутри окна, то она меняется местами с конечной точкой.

5. Анализируется код начальной точки для определения стороны окна с которой есть пересечение и выполняется расчет пересечения. При этом вычисленная точка пересечения заменяет начальную точку.

6. Определение нового кода начальной точки.

Эта схема реализована в процедуре V_CSclip, приведенной в Приложении 7.

0.6.2 Двумерный FC-алгоритм

В 1987 г. Собков, Поспишил и Янг [37] предложили алгоритм, названный ими FC-алгоритмом (Fast Clipping), также использующий кодирование, но не конечных точек, а линий целиком. Приведенное далее изложение алгоритма следует статье [37].

Схема кодирования близка к используемой в алгоритме Коэна-Сазерленда (рис. 0.6.2). Пространство разбивается на 9 неперекрывающихся областей, пронумерованных арабскими цифрами от 1 до 9. Коды, назначаемые концам отрезков, попавших в ту или иную область, приведены в двоичном и шестнадцатиричном виде (запись вида 0xD).

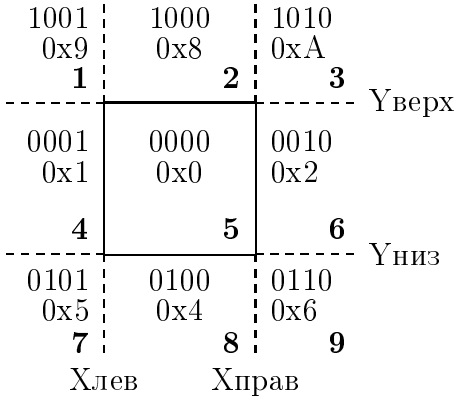


Рис. 0.6.2: Задание кодов для FC-алгоритма

Отрезок видим только в области 5, т.е. отрезок, координаты которого удовлетворяют условиям:

$$X \le X \le X \quad Y \le Y \le Y.$$

Каждая конечная точка отрезка V_0V_1 окажется с одной из этих областей. Комбинация кодов концов отрезка, называемая кодом линии, используется для определения возможных вариантов расположения отрезка и, следовательно, отсечения. Код линии формируется из кодов концов отрезка следующим образом:

$$\mathrm{LineCode}(V_0, V_1) \;=\; (\mathrm{Code}(V_0) \times 16) + \mathrm{Code}(V_1),$$

здесь $\mathrm{Code}(V_1)$ обозначает код конечной точки V_1,
$\mathrm{Code}(V_0) \times 16$ означает сдвиг кода начальной точки V_0 влево на 4 разряда.

Так как каждый код может принимать одно из 9 значений, то всего имеется 81 возможный вариант расположения отрезка. Но, если $\mathrm{Code}(V_0)$ равен $\mathrm{Code}(V_1)$, то $\mathrm{LineCode}(V_0,V_1)$ равен $\mathrm{LineCode}(V_1,V_0)$. Имеется всего 9 таких случаев: 1–1, 2–2, ... 9–9. Следовательно, число различных случаев уменьшается до 72.

Каждый LineCode требует своего набора вычислений для определения отсечения отрезка за минимальное время. Всего имеется 8 основных случаев отсечения, а остальные симметричны к ним. Рассмотрим эти 8 основных случаев. При этом будут использоваться следующие обозначения:

• начальная точка отрезка считается точкой номер 0 (V_0),

- конечная точка отрезка считается точкой номер 1 (V_1),
- ClipA_B обозначает алгоритм расчета перемещения конечной точки номер A на сторону окна B (расчет пересечения прямой линии, на которой расположен отсекаемый отрезок со стороной окна B).

Иллюстрации к случаям 1–7 приведены на рис. 0.6.3, для случая 8 — на рис. 0.6.4.

1. Начальная и конечная точки отрезка обе в области 5 (отрезок JK). Это простой случай принятия отрезка.

2. Начальная и конечная точки отрезка обе в области 4 (отрезок LA). Отрезок не пересекает видимую область, так что это простой случай отбрасывания.

3. Начальная точка в области 4, конечная — в области 1 (отрезок LB). Отрезок не пересекает видимую область, так что это простой случай отбрасывания.

4. Начальная точка в области 4, конечная — в области 2 (отрезки LC и LD). Отрезки явно пересекает Хлев, так что вначале надо определить соответствующую координату, используя алгоритм Clip0_Xleft. Для отрезка LC это дает $V_0y >$ Yверх, так что отрезок должен быть отброшен без дальнейших вычислений. Отрезок LD входит в окно с левой стороны и может выходить через верх. Следовательно, следующее отсечение должно быть Clip1_Top, после которого отрезок принимается.

5. Начальная точка в области 4, конечная — в области 3 (отрезки LE, LF и LG). Отрезки явно пересекает Хлев. Так же как и для случая 4 вначале применяется Clip0_Xleft и отрезок LE отбрасывается если $V_0y >$ Yверх. Если же получаем $V_0y \leq$ Yверх, то отрезок должен выйти из области видимости через верхнее или правое ребро. Применяем отсечение Clip1_Top и сравниваем новое значение X-координаты конечной точки — V_1x с Хправ. Если $V_1x \leq$ Хправ, то отрезок (LF) проходит через верхнюю сторону, отрезок принимается и дальнейшие вычисления не нужны. Иначе отрезок (LG) проходит через правую сторону и требуется отсечение Clip1_Right. Отсечение закончено, отрезок принимается.

6. Начальная точка в области 4, конечная — в области 6 (отрезок LH). Данный отрезок видим. Вначале используем Clip0_Xleft затем Clip1_Right и принимаем отрезок.

7. Начальная точка в области 4, конечная — в области 5 (отрезок LI). Данный отрезок видим. Просто используем Clip0_Xleft и принимаем отрезок.

8. Начальная точка V_0 (R, S, T или U) в области 7, конечная точка V_1 (W, X, Y или Z) — в области 3 (см. рис.0.6.4). В этом случае могут быть отброшены только два типа отрезков. Для минимизации вычислений используем Clip0_Xleft. Если $V_0y >$ Yверх, то это первый случай отбрасывания (отрезок RW). Clip1_Xright и проверка $V_1y <$ Yниз задают второй случай отбрасывания (отрезок UZ). Все другие отрезки должны быть видимы. Если $V_0y <$ Yниз, тогда $V_0 =$ T, иначе $V_0 =$ S. Если $V_0y <$ Yниз, то Clip1_Ybottom даст точку V_0 на ребре окна. Аналогично, если $V_1y >$ Yверх, то V_1=X и здесь требуется Clip1_Ytop перед приемом отрезка. Если $V_1y <$ Yверх, тогда $V_1 =$ Y.

Из этих восьми случаев легко симметрично сгенерировать все остальные.

Главное отличие FC-алгоритма от алгоритма Коэна-Сазерленда состоит в упорядочивании действий по отсечению. Эффективность алгоритма Коэна-Сазерленда ограничивается последовательным характером и фиксированным порядком действий по отсечению. Как пример (см. рис. 0.6.4) отрезок RW будет отсекаться в порядке: сверху, снизу, справа и слева. Число же отсечений для определения видимости равно 2 — снизу и слева. В FC-алгоритме, напротив, для каждого значения LineCode имеется свой набор действий по отсечению. Для приведенного выше примера потребуется только одно отсечение для определения невидимости отрезка

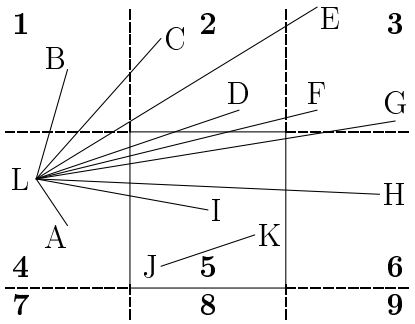


Рис. 0.6.3: Варианты расположения отрезка для неугловых областей

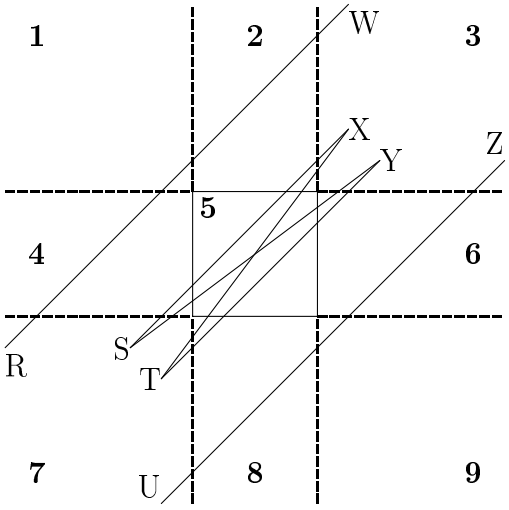


Рис. 0.6.4: Случай угловых областей

RW. Кроме этого, повышению эффективности FC-алгоритма по сравнению с CS-алгоритмом способствует отсутствие ненужных циклов и, следовательно, перевычислений кодов конечных точек.

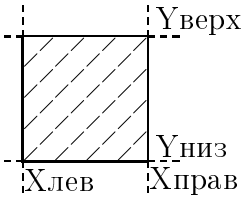
В Приложении 7 приведена C-подпрограмма V_FCclip, реализующая FC-алгоритм и свободная от ошибок в подпрограмме, приведенной в [37]. Можно заметно сократить объем ее программного кода учтя симметрию и использовав указатели на данные либо переставляя данные. Например, в подпрограмме V_FCclip для отрезка LH (см. рис. 0.6.3, если он идет слева-направо вначале выполняется отсечение для начальной точки по левой стороне окна и затем для конечной - по правой. Если же отрезок идет справа-налево, то вначале вычисляется отсечение начальной точки по правой стороне и затем конечной — по левой. Очевидно, что эти два случая идентичны если поменять местами координаты начальной и конечной точек.

0.6.3 Двумерный алгоритм Лианга-Барски

В 1982 г. Лианг и Барски [32] предложили алгоритмы отсечения прямоугольным окном с использованием параметрического представления для двух, трех и четырехмерного отсечения. По утверждению авторов, данный алгоритм в целом превосходит алгоритм Коэна-Сазерленда.

Однако в работе [37] показывается, что это утверждение справедливо только для случая когда оба конца видимого отрезка вне окна и окно небольшое (до 50×50 при разрешении 1000×1000). Приведенное далее изложение двумерного варианта алгоритма следует, основном, работе [32].

Как уже говорилось, при 2D отсечении прямые отсекаются по 2D области, называемой окном отсечения. В частности, внутренняя часть окна отсечения может быть выражена с помощью следующих неравенств (рис. 0.6.5).



$$\begin{array}{ccccc} X & \leq & x & \leq & X \\ Y & \leq & y & \leq & Y \end{array} \qquad (0.6.1)$$

Рис. 0.6.5: Внутренняя часть окна отсечения

Продолжим каждую из четырех границ окна до бесконечных прямых. Каждая из таких прямых делит плоскость на 2 области. Назовем "видимой частью" ту, в которой находится окно отсечения, как это показано на рис. 0.6.6. Видимой части соответствует внутренняя сторона линии границы. Невидимой части плоскости соответствует внешняя сторона линии границы.

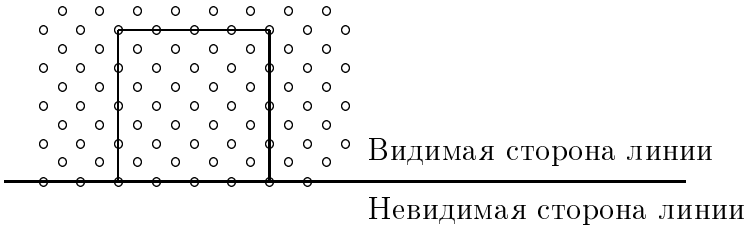


Рис. 0.6.6: Видимая часть линии границы

Таким образом, окно отсечения может быть определено как область, которая находится на внутренней стороне всех линий границ.

Отсекаемый отрезок прямой может быть преобразован в параметрическое представление следующим образом. Пусть конечные точки отрезка есть V_0 и V_1 с координатами (x0,y0) и (x1,y1), соответственно. Тогда параметрическое представление линии может быть задано следующим образом:

$$x = x0 + dx \cdot t; \quad y = y0 + dy \cdot t, \qquad (0.6.2)$$

$$dx = x1 - x0; \quad dy = y1 - y0. \qquad (0.6.3)$$

Или в общем виде для отрезка, заданного точками V_0 и V_1:

$$V(t) = V_0 + (V_1 - V_0) \cdot t \qquad (0.6.4)$$

Для точек V_0 и V_1 параметр t равен 0 и 1, соответственно. Меняя t от 0 до 1 перемещаемся по отрезку V_0V_1 от точки V_0 к точке V_1. Изменяя t в интервале от $-\infty$ до $+\infty$, получаем бесконечную (далее удлиненную) прямую, ориентация которой — от точки V_0 к точке V_1.

0 1

0 1 2 3 2 3

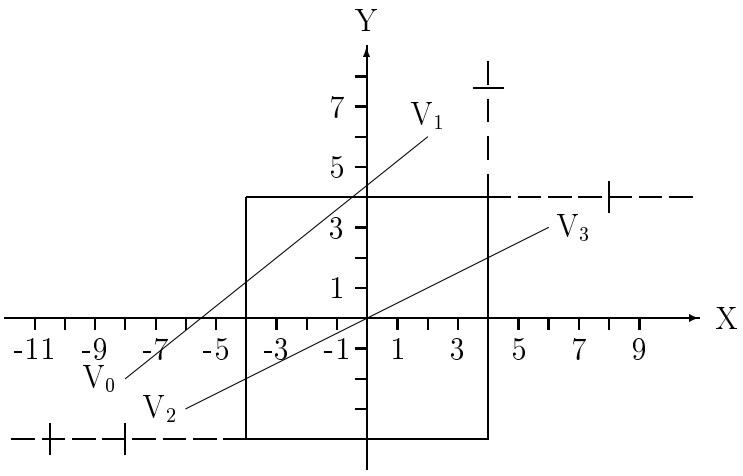


Рис. 0.6.7: Пример расчета отсечения

0 1

$$y = -2 + 8t; \quad x = -8 + 10t.$$

V_2V_3

$$y = -3 + 6t; \quad x = -6 + 12t.$$

0

0 1 2 3

0 1

2 3

1

$_0$ $_1$ $_0 \leq _1$

Таблица 0.6.1: Расчет отсечения для отрезка V_0V_1

	Ребро окна	t	t0, t1	X	Y
Внутрь	Левое	4/10	t0= tmax= 4/10	-4	1.2
	Нижнее	-2/8	вне [0,1]		
Наружу	Правое	12/10	вне [0,1]		
	Верхнее	6/8	t1= tmin= 6/8	-0.5	4

Таблица 0.6.2: Расчет отсечения для отрезка V_2V_3

	Ребро окна	t	t0, t1	X	Y
Внутрь	Левое	2/12	t0= tmax= 2/12	-4	-2
	Нижнее	-1/6	вне [0,1]		
Наружу	Правое	10/12	t1= tmin=10/12		
	Верхнее	7/6	вне [0,1]	4	2

Однако вернемся к формальному рассмотрению алгоритма отсечения.

Подставляя параметрическое представление, заданное уравнениями (0.6.2) и (0.6.3), в неравенства (0.6.1), получим следующие соотношения для частей удлиненной линии, которая находится в окне отсечения:

$$\begin{array}{llll} -dx \cdot t \leq x0 - X & & dx \cdot t \leq X - x0, \\ -dy \cdot t \leq y0 - Y & & dy \cdot t \leq Y - y0. \end{array} \tag{0.6.5}$$

Заметим, что соотношения (0.6.5) — неравенства, описывающие внутреннюю часть окна отсечения, в то время как равенства определяют его границы.

Рассматривая неравенства (0.6.5), видим, что они имеют одинаковую форму вида:

$$P_i \cdot t \leq Q_i \quad i = 1, 2, 3, 4. \tag{0.6.6}$$

Здесь использованы следующие обозначения:

$$\begin{array}{llll} P_1 = -dx; & & Q_1 = x0 - X; \\ P_2 = dx; & & Q_2 = X - x0; \\ P_3 = -dy; & & Q_3 = y0 - Y; \\ P_4 = dy; & & Q_4 = Y - y0. \end{array} \tag{0.6.7}$$

Вспоминая определения внутренней и внешней стороны линии границы (см. рис. 0.6.6), замечаем, что каждое из неравенств (0.6.6) соответствует одной из граничных линий (левой,

правой, нижней и верхней, соответственно) и описывает ее видимую сторону. (Например, для i=1 имеем: $P_1 \cdot t \leq Q_1 \Longrightarrow -dx \cdot t \leq x0 - X \Longrightarrow x0 + dx \cdot t \geq X$). Удлиним V_0V_1 в бесконечную прямую. Тогда каждое неравенство задает диапазон значений параметра t, для которых эта удлиненная линия находится на видимой стороне соответствующей линии границы. Более того, конкретное значение параметра t для точки пересечения есть $t = Q_i/P_i$. Причем знак Q_i показывает на какой стороне соответствующей линии границы находится точка V_0. А именно, если $Qi \geq 0$, тогда V_0 находится на видимой стороне линии границы, включая и ее. Если же $Q_i < 0$, тогда V_0 находится на невидимой стороне.

Рассмотрим P_i в соотношениях (0.6.7). Ясно, что любое P_i может быть меньше 0, больше 0 и равно 0.

$P_i < 0$

Если $P_i < 0$, тогда соответствующее неравенство становится:

$$t \geq Q_i \,/\, P_i. \tag{0.6.8}$$

Для пояснения на рис. 0.6.8 показано пересечение с левой и правой границами при $P_i < 0$.

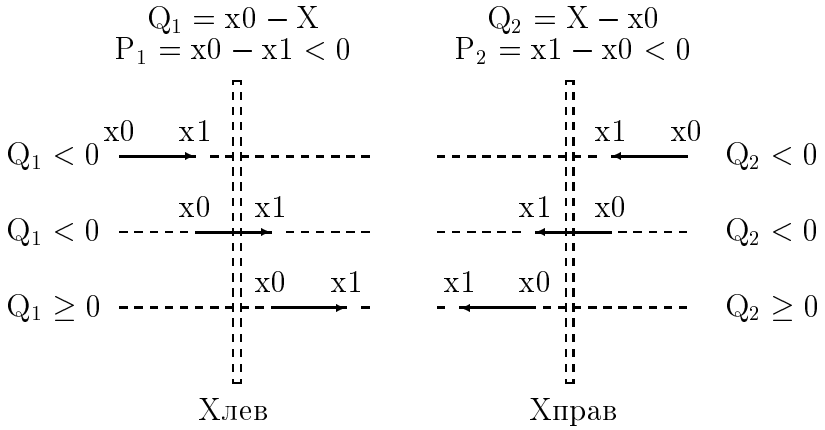


Рис. 0.6.8: Пересечение удлиненной линии, определяемой точками V_0V_1 и идущей с невидимой на видимую сторону, с левой и правой границами.

Очевидно, что диапазон значений параметра t, для которых удлиненная линия находится на видимой стороне соответствующей граничной линии, имеет минимум в точке пересечения направленной удлиненной линии, заданной вектором V_0V_1 и идущей с невидимой на видимую сторону граничной линии (так как только на границе t равно $Q_i \,/\, P_i$, а в остальной части видимой стороны больше).

$Pi > 0$

Аналогично, если $P_i > 0$, тогда соответствующее неравенство становится:

$$t \leq Q_i \,/\, P_i. \tag{0.6.9}$$

Для пояснения на рис. 0.6.9 показано пересечение с левой и правой границами при $P_i > 0$.

Так как значения параметра t только на границе равны Q_i/P_i, а в остальной видимой части меньше Q_i/P_i, то значение параметра t имеет максимум на границе.

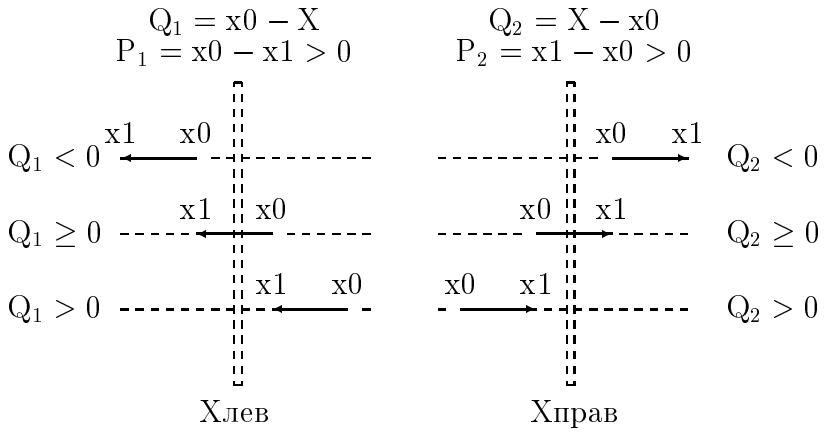


Рис. 0.6.9: Пересечение удлиненной линии, определяемой точками V_0V_1 и идущей с видимой на невидимую сторону, с левой и правой границами.

$\mathbf{P_i = 0}$

Наконец, если $P_i = 0$, тогда соответствующее неравенство превращается в:

$$0 \leq Q_i. \tag{0.6.10}$$

Заметим, что здесь нет зависимости от t, т.е. неравенство выполняется для всех t, если $Q_i \geq 0$ и не имеет решения при $Q_i < 0$. Для пояснения на рис. 0.6.10 иллюстрируется случай $P_i = 0$.

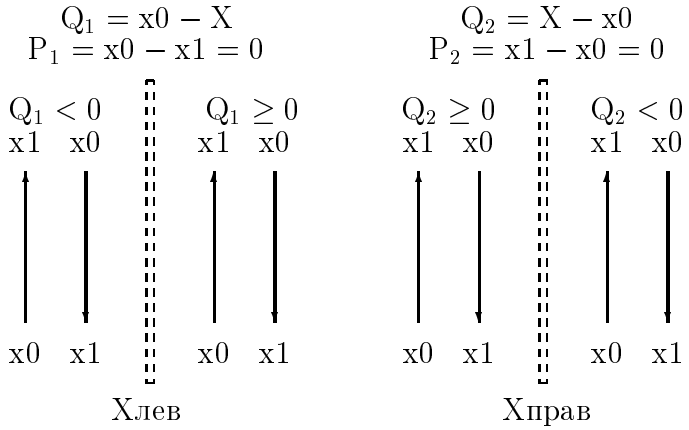


Рис. 0.6.10: Относительное расположение удлиненной линии, заданной точками V_0V_1 и идущей параллельно левой и правой границам.

Геометрически, если $P_i = 0$, то нет точек пересечения удлиненной линии, определяемой точками V_0V_1, с линиями границы. Более того, если $Q_i < 0$, то удлиненная линия находится на внешней стороне линии границы, а при $Q_i \geq 0$ находится на внутренней стороне (включая ее). В последнем случае отрезок V_0V_1 может быть видим или нет в зависимости от того где

находятся точки V_0V_1 на удлиненной линии. В предыдущем же случае нет видимого сегмента, так как удлиненная линия вне окна, т.е. это случай тривиального отбрасывания.

Все эти случаи суммированы на блок-схеме, представленной на рис. 0.6.11.

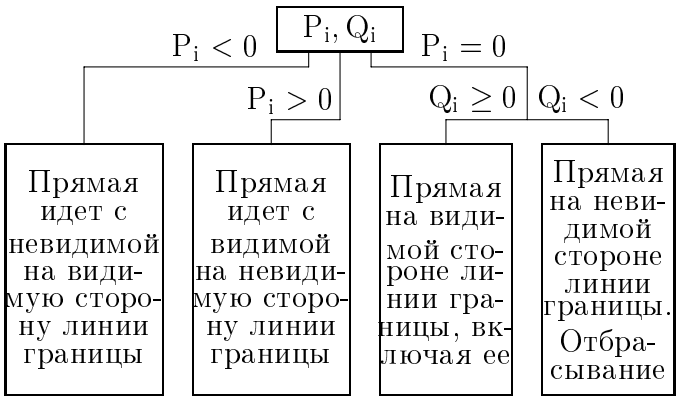


Рис. 0.6.11: Блок-схема алгоритма Лианга-Барски

Итак, рассмотрение четырех неравенств дает диапазон значений параметра t, для которого удлиненная линия находится внутри окна отсечения. Однако, отрезок V_0V_1 только часть удлиненной линии и он описывается значениями параметра t в диапазоне: $0 \leq t \leq 1$. Таким образом, решение задачи двумерного отсечения эквивалентно решению неравенств (0.6.6) при условии $0 \leq t \leq 1$. Решение этой задачи сводится к далее описанному отысканию максимумов и минимумов.

Вспомним, что для всех i таких, что $P_i < 0$, условие видимости имеет вид: $t \geq Q_i / P_i$. Из условия принадлежности точек удлиненной линии отрезку V_0V_1 имеем $t \geq 0$. Таким образом, нужно искать:

$$t \geq \max (\{ Q_i / P_i \mid P_i < 0,\ i = 1,2,3,4 \} \cup \{0\}\}. \tag{0.6.11}$$

Аналогично, для всех i таких что $P_i > 0$, условие видимости — $t \leq Q_i / P_i$ и, следовательно, ≤ 1.

$$t \leq \min (\{ Q_i / P_i \mid P_i > 0,\ i = 1,2,3,4 \} \cup \{1\}\}. \tag{0.6.12}$$

Наконец, для всех i, таких что $P_i = 0$ следует проверить знак Q_i. Если $Q_i < 0$, то это случай тривиального отбрасывания, задача отсечения решена и дальнейшие вычисления не нужны. Если же $Q_i \geq 0$, то информации, даваемой неравенством, недостаточно и это неравенство игнорируется.

Правая часть неравенств (0.6.11) и (0.6.12) — значения параметра t, соответствующие началу и концу видимого сегмента, соответственно. Обозначим эти значения как t_0 и t_1:

$$\begin{array}{rcl} t_0 & \geq & \max(\{Q_i/P_i | P_i < 0, \quad i = 1,2,3,4\} \quad \cup\{0\}\}, \\ t_1 & \leq & \min(\{Q_i/P_i | P_i > 0, \quad i = 1,2,3,4\} \quad \cup\{1\}\}. \end{array} \tag{0.6.13}$$

Если сегмент отрезка V_0V_1 видим, то ему соответствует интервал параметра:

$$t_0 \leq t \leq t_1. \tag{0.6.14}$$

Следовательно, необходимое условие видимости сегмента:

$$t_0 \leq t_1 \tag{0.6.15}$$

Но это недостаточное условие, так как оно игнорирует случай тривиального отбрасывания при $P_i = 0$, если $Q_i < 0$. Тем не менее это достаточное условие для отбрасывания, т.е. если $t_0 > t_1$, то отрезок должен быть отброшен. Алгоритм проверяет, если $P_i = 0$ с $Q_i < 0$, или $t_0 > t_1$ и в этом случае отрезок немедленно отбрасывается без дальнейших вычислений.

В алгоритме t_0 и t_1 инициализируются в 0 и 1, соответственно. Затем последовательно рассматривается каждое отношение Q_i/P_i.

Если $P_i < 0$, то отношение вначале сравнивается с t_1 и, если оно больше t_1, то это случай отбрасывания. В противном случае оно сравнивается с t_0 и, если оно больше, то t_0 должно быть заменено на новое значение.

Если $P_i > 0$, то отношение вначале сравнивается с t_0 и, если оно меньше t_0, то это случай отбрасывания. В противном случае оно сравнивается с t_1 и, если оно меньше, то t_1 должно быть заменено на новое значение.

Наконец, если Pi $= 0$ $Q_i < 0$, то это случай отбрасывания.

На последнем этапе алгоритма, если отрезок еще не отброшен, то t_0 и t_1 используются для вычисления соответствующих точек. Однако, если $t_0 = 0$, то конечная точка равна V_0 и не требуется вычислений. Аналогично, если $t_1 = 1$, то конечная точка — V_1 и вычисления также не нужны.

Геометрический смысл этого процесса состоит в том, что отрезок удлиняется для определения где эта удлиненная линия пересекает каждую линию границы. Более детально, каждая конечная точка заданного отрезка V_0V_1 используется как начальное значение для конечных точек отсеченного отрезка C_0C_1. Затем вычисляются точки пересечения удлиненной линии с каждой линией границы (эти вычисления соответствуют вызову процедуры LB_tclip в программе). Если для данной линии границы направление, определяемое V_0V_1, идет с невидимой на видимую сторону линии границы, то эта точка пересечения вначале сравнивается с C_1. Если точка находится далее вдоль линии, тогда C_1 (и таким образом, $_{01}$) должна быть на невидимой стороне линии, поэтому отрезок должен быть отброшен. В противном случае точка пересечения сравнивается с C_0; если точка далее вдоль линии, тогда C_0 перемещается вперед к этой точке.

С другой стороны, если направление с видимой на невидимую сторону, тогда точка пересечения вначале сравнивается с C_0. Если C_0 далее вдоль линии, чем точка пересечения, тогда C_0 (и, следовательно C_0C_1) находится на невидимой стороне линии границы, т.е. отрезок должен быть отброшен. В противном случае точка пересечения сравнивается с C_1 и, если C_1 далее вдоль линии, тогда C_1 перемещается назад к точке пересечения.

Наконец, если удлиненная линия параллельна граничной линии и она на невидимой стороне, то отрезок отбрасывается. В конце алгоритма, если отрезок не отброшен, тогда C_0 и C_1 используются как конечные точки видимой части отрезка.

В Приложении 7 приведена C-подпрограмма V_LBclip, реализующая описанный выше алгоритм.

0.6.4 Двумерный алгоритм Кируса-Бека

Все рассмотренные выше алгоритмы проводили отсечение по прямоугольному окну, стороны которого параллельны осям координат. Это, конечно, наиболее частый случай отсечения.

Однако, во многих случаях требуется отсечение по произвольному многоугольнику, например, в алгоритмах удаления невидимых частей сцены. В этом случае наиболее удобно использование параметрического представления линий, не зависящего от выбора системы координат.

Из предыдущего пункта ясно, что для выполнения отсечения в параметрическом представлении необходимо иметь способ определения ориентации удлиненной линии, содержащей отсекаемый отрезок, относительно линии границы — с внешней стороны на внутреннюю или с внутренней на внешнюю, а также иметь способ определения расположения точки, принадлежащей отрезку, относительно окна — вне, на границе, внутри.

Для этих целей в алгоритме Кируса-Бека [29], реализующем отсечение произвольным выпуклым многоугольником, используется вектор внутренней нормали к ребру окна.

Внутренней нормалью $\mathbf{N}$ в точке A к стороне окна называется нормаль, направленная в сторону области, задаваемой окном отсечения.

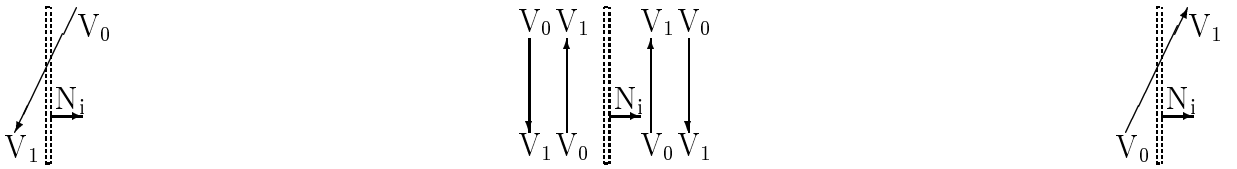
Рассмотрим основные идеи алгоритма Кируса-Бека.

Так как многоугольник предполагается выпуклым, то может быть только две точки пересечения отрезка с окном. Поэтому надо найти два значения параметра t, соответствующие начальной и конечной точкам видимой части отрезка.

Пусть $\mathbf{N}_i$ — внутренняя нормаль к i-й граничной линии окна, а $\mathbf{P} = \mathbf{V}_1 - \mathbf{V}_0$ — вектор, определяющий ориентацию отсекаемого отрезка, тогда ориентация отрезка относительно i-й стороны окна определяется знаком скалярного произведения $P_i = \mathbf{N}_i \cdot \mathbf{V}$, равного произведению длин векторов на косинус наименьшего угла, требуемого для поворота вектора $\mathbf{N}_i$ до совпадения по направлению с вектором $\mathbf{V}$:

$$P_i = \mathbf{N}_i \cdot \mathbf{P} = \mathbf{N}_i \cdot (\mathbf{V}_1 - \mathbf{V}_0). \tag{0.6.16}$$

$$\begin{array}{ll} P_i < 0 & \text{отсекаемый отрезок направлен с внутренней на внешнюю стороны i-й} \\ & \text{граничной линии окна (см. рис. 0.6.12а).} \\ P_i = 0 & \text{точки } V_0 \text{ и } V_1 \text{ либо совпадают, либо отсекаемый отрезок параллелен} \\ & \text{i-й граничной линии окна (см. рис. 0.6.12б).} \\ P_i > 0 & \text{отсекаемый отрезок направлен с внешней на внутреннюю сторону i-й} \\ & \text{граничной линии окна (см. рис. 0.6.12в).} \end{array} \tag{0.6.17}$$



а) Изнутри наружу

б) Параллельно границе

в) Снаружи внутрь

Рис. 0.6.12: Ориентация отсекаемого отрезка относительно окна

Для определения расположения точки относительно окна вспомним параметрическое представление отсекаемого отрезка:

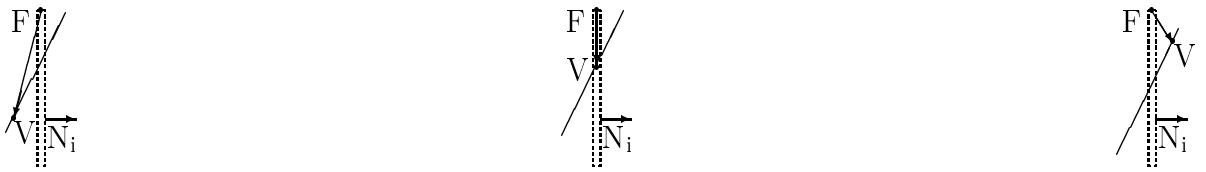
$$V(t) = V_0 + (V_1 - V_0) \cdot t; \quad 0 \geq t \geq 1. \tag{0.6.18}$$

Рассмотрим теперь скалярное произведение внутренней нормали $\mathbf{N}_i$ к i-й границе на вектор $\mathbf{Q}(t) = \mathbf{V}(t) - \mathbf{F}_i$, начинающийся в начальной точке ребра окна и заканчивающийся в некоторой точке V(t) удлиненной линии.

$$Q_i \;=\; \mathbf{N}_i \cdot \mathbf{Q} \;=\; \mathbf{N}_i \cdot [\mathbf{V}(t) \;-\; \mathbf{F}_i] \qquad i \;=\; 1, 2, 3 \;\ldots \tag{0.6.19}$$

Аналогично предыдущему имеем (рис. 0.6.13):

$$\begin{array}{ll} Q_i < 0 & \mathbf{V}(t) \\ Q_i = 0 & \mathbf{V}(t) \\ Q_i > 0 & \mathbf{V}(t) \end{array} \tag{0.6.20}$$



а) Точка V вне

б) Точка V на границе

в) Точка V внутри

Рис. 0.6.13: Расположение точки относительно окна

Подставляя в (0.6.19) параметрическое представление (0.6.18), получим условие пересечения отрезка с границей окна:

$$\mathbf{N}_i \cdot [\; \mathbf{V}_0 \;+\; (\mathbf{V}_1 \;-\; \mathbf{V}_0) \cdot t \;-\; \mathbf{F}_i \;] \;=\; 0 \tag{0.6.21}$$

Раскрывая скобки, получим:

$$\mathbf{N}_i \cdot [\; \mathbf{V}_0 \;-\; \mathbf{F}_i \;] \;+\; \mathbf{N}_i \cdot [\; \mathbf{V}_1 \;-\; \mathbf{V}_0 \;] \cdot t \;=\; 0. \tag{0.6.22}$$

Используя (0.6.16) и (0.6.19) перепишем (0.6.21):

$$(\; \mathbf{N}_i \;\cdot\; \mathbf{P} \;) \;\cdot\; t \;+\; \mathbf{N}_i \;\cdot\; \mathbf{Q} \;=\; P_i \;\cdot\; t \;+\; Q_i. \tag{0.6.23}$$

Разрешая (0.6.22) относительно t, получим:

$$t \;=\; -\frac{Q_i}{P_i} \;=\; -\frac{\mathbf{N}_i \cdot \mathbf{Q}}{\mathbf{N}_i \cdot \mathbf{P}} \qquad P_i \neq 0, \;\; i \;=\; 1, 2, 3, \ldots \tag{0.6.24}$$

Это уравнение и используется для вычисления значений параметров, соответствующих начальной и конечной точкам видимой части отрезка.

Как следует из (0.6.17), P_i равно нулю если отрезок либо вырожден в точку, либо параллелен границе. В этом случае следует проанализировать знак Q_i и принять или не принять решение об отбрасывании отрезка целиком в соответствии с условиями (0.6.17).

Если же P_i не равно 0, то уравнение (0.6.24) используется для вычисления значений параметров t, соответствующих точкам пересечений удлиненной линии с линиями границ.

Алгоритм построен следующим образом:

Искомые значения параметров t_0 и t_1 точек пересечения инициализируются значениями 0 и 1, соответствующими началу и концу отсекаемого отрезка.

Затем в цикле для каждой i-й стороны окна отсечения вычисляются значения скалярных произведений, входящих в (0.6.23).

Если очередное P_i равно 0, то отсекаемый отрезок либо вырожден в точку, либо параллелен i-й стороне окна. При этом достаточно проанализировать знак Q_i. Если $Q_i < 0$, то отрезок вне окна и отсечение закончено иначе рассматривается следующая сторона окна.

Если же P_i не равно 0, то по (0.6.24) можно вычислить значение параметра t для точки пересечения отсекаемого отрезка с i-й границей. Так как отрезок V_0V_1 соответствует диапазону $0 \le t \le 1$, то все решения, выходящие за данный диапазон следует отбросить. Выбор оставшихся решений определяется знаком P_i.

Если $P_i < 0$, т.е. удлиненная линия направлена с внутренней на внешнюю стороны граничной линии, то ищутся значения параметра для конечной точки видимой части отрезка. В этом случае определяется минимальное значение из всех получаемых решений. Оно даст значение параметра t_1 для конечной точки отсеченного отрезка. Если текущее полученное значение t_1 окажется меньше, чем t_0, то отрезок отбрасывается, так как нарушено условие $t_0 \le t_1$.

Если же $P_i > 0$, т.е. удлиненная линия направлена с внешней на внутреннюю стороны граничной линии, то ищутся значения параметра для начальной точки видимой части отрезка. В этом случае определяется максимальное значение из всех получаемых решений. Оно даст значение параметра t_0 для начальной точки отсеченного отрезка. Если текущее полученное значение t_0 окажется больше, чем t_1, то отрезок отбрасывается, так как нарушено условие $t_0 \le t_1$.

На заключительном этапе алгоритма значения t_0 и t_1 используются для вычисления координат точек пересечения отрезка с окном. При этом, если $t_0 = 0$, то начальная точка осталась V_0 и вычисления не нужны. Аналогично, если $t_1 = 1$, то конечная точка осталась V_1 и вычисления также не нужны.

Все эти случаи пояснены на блок-схеме, представленной на рис. 0.6.14.

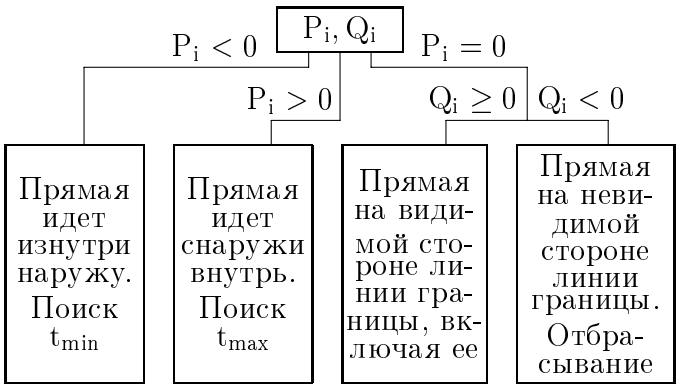


Рис. 0.6.14: Блок-схема алгоритма Кируса-Бека

Вычисления значений параметров t_0 и t_1 выполняются в соответствии с выражениями (0.6.25).

$$\begin{aligned} t_0 &\ge \max(\{-Q_i/P_i | P_i > 0, \quad i = 1, 2, \ldots\} \cup \{0\}), \\ t_1 &\le \min(\{-Q_i/P_i | P_i < 0, \quad i = 1, 2, \ldots\} \cup \{1\}). \end{aligned} \tag{0.6.25}$$

В Приложении 7 приведена C-подпрограмма V_CBclip, реализующая описанный выше алгоритм.

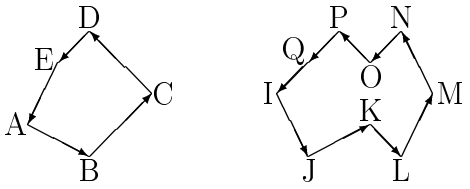
Проверка выпуклости и определение нормалей

Как видно из описания, алгоритм Кируса-Бека отсекает только по выпуклому окну. Кроме этого требуются значения внутренних нормалей к сторонам окна. Естественно выполнить эти вычисления в момент задания окна, так как следует ожидать, что одним окном будет отсекаться достаточно много отрезков.

Алгоритм с использованием векторных произведений

Проверка на выпуклость может производиться анализом знаков векторных произведений смежных ребер (рис. 0.6.15).

$$|\vec{A} \times \vec{B}| = A \cdot B \cdot \sin\,(\overset{\wedge}{AB})$$



а) выпуклый б) невыпуклый

Рис. 0.6.15: Проверка выпуклости и определение нормалей

Если знак векторного произведения равен 0, то вершина вырождена, т.е. смежные ребра лежат на одной прямой (см. рис. 0.6.15 б), вершина Q).

Если все знаки равны 0, то многоугольник отсечения вырождается в отрезок.

Если же векторные произведения имеют разные знаки, то многоугольник отсечения невыпуклый (см. рис. 0.6.15 б)).

Если все знаки неотрицательные, то многоугольник выпуклый, причем обход вершин выполняется против часовой стрелки (см. рис. 0.6.15 а)), т.е. внутренние нормали ориентированы влево от контура. Следовательно вектор внутреннего перпендикуляра к стороне может быть получен поворотом ребра на $+90^\circ$ (в реализации алгоритма вычисления нормалей на самом деле вычисляется не нормаль к стороне, а перпендикуляр, так как при вычислении значения t по соотношению (0.6.22) длина не важна).

Если все знаки неположительные, то многоугольник выпуклый, причем обход вершин выполняется по часовой стрелке, т.е. внутренние нормали ориентированы вправо от контура. Следовательно вектор внутреннего перпендикуляра к стороне может быть получен поворотом ребра на -90°.

Описанный алгоритм реализован в процедуре V_SetPclip, приведенной в Приложении 7 и предназначенной для установки многоугольного окна отсечения.

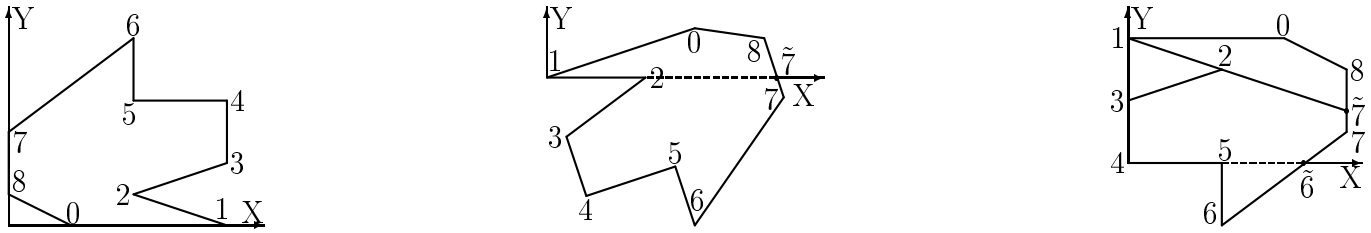
Разбиение невыпуклых многоугольников

Одновременное проведение операций проверки на выпуклость и разбиение простого невыпуклого многоугольника на выпуклые обеспечивается методом переноса и поворотов окна.

Алгоритм метода при обходе вершин многоугольника против часовой стрелки состоит в следующем:

1. Для каждой i-й вершины многоугольник сдвигается для переноса упомянутой вершины в начало координат.
2. Многоугольник поворачивается против часовой стрелки для совмещения (i+1)-й вершины с положительной полуосью X.
   Вектор внутреннего перпендикуляра к ребру, образованному вершинами i-й и (i+1)-й, вычисляется поворотом ребра на -90° против часовой стрелки.
3. Анализируется знак Y-координаты (i+2)-й вершины.
   Если $Y_{i+2} \geq 0$, то в (i+1)-й вершине выпуклость.
   Если $Y_{i+2} \geq 0$, то в (i+1)-й вершине невыпуклость.
   Если имеется невыпуклость, то многоугольник разрезается на два вдоль положительной полуоси X.
   Для этого вычисляется пересечение положительной полуоси X с первой из сторон. Формируются два новых многоугольника: первый многоугольник — вершины с (i+1)-й до точки пересечения — вершины 2, 3, 4, 6, 7, $\tilde{7}$ на рис. 0.6.16 б);
   второй многоугольник — все остальные вершины — вершины $\tilde{7}$, 8, 0, 1 на рис. 0.6.16 б)

Так как вновь полученные многоугольники могут в свою очередь оказаться невыпуклыми, алгоритм применяется к ним, пока все многоугольники не станут выпуклыми.



а) Исходное окно

б) Невыпуклость после вершины 2

в) Невыпуклость после вершины 5

Рис. 0.6.16: Проверка выпуклости и разбиение многоугольника

Повторное применение алгоритма в многоугольнику, образованному вершинами 2, 3, 4, 6, 7, $\tilde{7}$, показано на рис. 0.6.16 в).

Данный алгоритм не обеспечивает минимальность числа вновь полученных выпуклых многоульников и некорректно работает если имеется самопересечение сторон, как это показано на рис. 0.6.17.

0.6.5 Сравнение алгоритмов двумерного отсечения

Во многих работах приводятся качественные соображения по быстродействию различных алгоритмов отсечения. В части работ, например, [32] или [37] приводятся результаты численных экспериментов по измерению скорости. Как правило, авторы работ этими экспериментами подтверждают преимущество своих алгоритмов.

В целом можно отметить несколько методических неточностей проведения таких экспериментов:

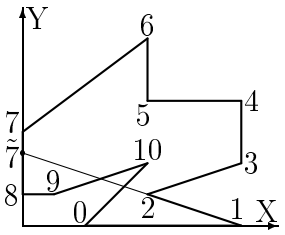


Рис. 0.6.17: Многоугольник с самопересечением сторон

- неясно насколько одинаково хороши реализации собственного и сравниваемых алгоритмов,
- эксперименты ([32] и [37] проводились в среде ОС UNIX и нет убедительных свидетельств отсутствия влияния окружения на результаты,
- неясно насколько правильно выбиралось число повторений одного отсечения относительно минимального измеряемого кванта времени.

Исходя из этих соображений были проведены численные эксперименты по измерению быстродействия алгоритмов отсечения Коэна-Сазерленда, FC-алгоритма, Лианга-Барски и Кируса-Бека.

Использовались подпрограммы, приведенные в Приложении 7 при отсечении окнами различных размеров при полном разрешении 1000×1000. Процедуры транслировались и исполнялись на 486/DX4/100 в среде на Turbo C под управлением MS DOS 6.22.

Аналогично [37] были подготовлены 5 наборов данных по 1000 отрезков каждый со случайной генерацией конечных точек при следующих ограничениях:

1. Обе конечные точки отрезка внутри окна.
2. Одна конечная точка отрезка в окне, другая вне.
3. Обе конечные точки вне окна но с видимым сегментом.
4. Обе конечные точки вне окна и отрезок невидим.
5. Обе конечные точки генерировались случайно без ограничений.

Сгенерированные данные сохранялись в файлах и считывались в оперативную память перед очередным прогоном теста. Процедуры отсечения использовали данные из оперативной памяти. Для исключения временных затрат, связанных с организацией циклов и запросом координат отрезков, предварительно прогонялся тест с использованием “пустой” процедуры отсечения. Отсечение каждого отрезка проводилось 1000 раз. Измерение времени проводилось перед началом цикла по координатам.

Результаты измерений приведены в таблицах 0.6.3, 0.6.4, 0.6.5, 0.6.6, 0.6.7. Первая колонка таблиц — мои измерения. Вторая колонка таблиц — данные из [37]. Последние проводились на DEC VAX 8600 с ускорителем плавающей арифметики, транслировались C-компилятором без оптимизации и исполнялись под управлением ULTRIX V 1.1 (C).

Таблица 0.6.3: Время (с) для простого приема отрезка.

Эксперимент на 486/DX4/100

Окно	CS	FC	LB	CB

Данные из статьи [37]

Окно	CS	FC	LB	CB
0.0	25.3	11.0	113.0	377.1
10.0	25.3	11.0	121.6	393.9
20.0	25.3	10.7	122.3	390.6
30.0	25.4	11.0	122.0	395.0
40.0	25.4	11.0	123.2	394.3
50.0	25.3	11.2	121.5	394.6
60.0	25.0	10.9	121.6	394.5
70.0	25.4	11.0	122.7	394.7
80.0	25.2	10.2	121.4	393.3
90.0	25.2	11.0	124.8	394.4
98.0	25.1	10.6	122.7	393.6

Таблица 0.6.4: Время (с) для отрезков с одним концом в окне.

Эксперимент на 486/DX4/100

Окно	CS	FC	LB	CB

Данные из статьи [37]

Окно	CS	FC	LB	CB
0.0	124.0	71.6	146.7	390.5
10.0	110.2	61.1	148.5	399.1
20.0	107.0	57.7	148.0	398.5
30.0	104.3	57.2	147.5	401.4
40.0	101.8	55.2	148.7	403.9
50.0	100.8	54.8	149.2	397.2
60.0	100.8	54.4	148.9	400.7
70.0	96.9	53.6	146.9	399.5
80.0	96.3	51.9	148.3	399.0
90.0	95.6	51.9	148.5	400.3
98.0	94.6	50.6	147.1	400.7

Таблица 0.6.5: Время (с) для видимых отрезков вне окна.

Эксперимент на 486/DX4/100

Окно	CS	FC	LB	CB

Данные из статьи [37]

Окно	CS	FC	LB	CB
0.0	233.0	135.4	174.8	406.6
10.0	193.1	104.9	173.2	406.4
20.0	183.0	98.4	174.8	406.1
30.0	177.5	97.5	174.4	409.4
40.0	177.1	96.6	175.6	409.8
50.0	173.9	95.5	174.2	405.0
60.0	168.9	93.4	173.2	405.9
70.0	168.0	92.9	173.4	406.5
80.0	166.4	92.1	173.6	413.8
90.0	164.8	91.7	173.3	411.9
98.0	163.9	91.0	172.6	403.7

Таблица 0.6.6: Время (с) для невидимых отрезков вне окна.

Эксперимент на 486/DX4/100

Окно	CS	FC	LB	CB

Данные из статьи [37]

Окно	CS	FC	LB	CB
0.0	46.9	25.0	75.9	230.4
10.0	36.9	18.2	79.2	233.9
20.0	34.7	16.8	79.3	237.0
30.0	31.9	15.1	78.8	234.6
40.0	29.1	13.4	78.7	225.9
50.0	28.4	12.6	78.8	229.0
60.0	26.4	11.7	78.0	226.1
70.0	25.5	11.7	77.2	223.1
80.0	24.7	11.2	76.2	222.3
90.0	25.0	10.8	75.9	220.7
98.0	24.6	10.8	68.9	197.5

Таблица 0.6.7: Время (с) для случайных отрезков.

Эксперимент на 486/DX4/100

Окно	CS	FC	LB	CB

Данные из статьи [37]

Окно	CS	FC	LB	CB
0.0	53.7	27.9	80.1	240.5
10.0	104.6	55.9	124.5	324.3
20.0	100.4	54.5	131.7	347.7
30.0	98.4	53.0	137.7	367.7
40.0	95.1	50.7	139.4	375.4
50.0	87.2	46.5	140.9	387.1
60.0	76.0	41.1	138.7	388.7
70.0	65.1	34.4	135.4	393.0
80.0	52.8	26.8	132.1	392.1
90.0	39.0	19.0	126.8	395.3
98.0	28.0	12.4	123.5	393.4

0.6.6 Трехмерное отсечение отрезка

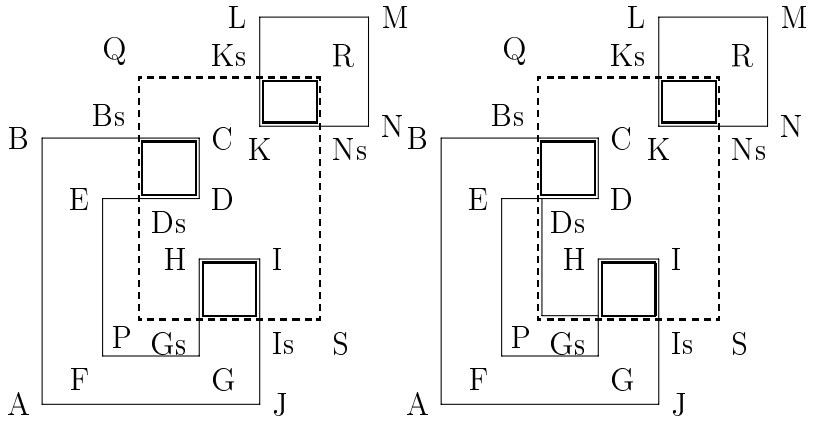
0.6.7 Отсечение отрезка в однородных координатах

0.7 ОТСЕЧЕНИЕ МНОГОУГОЛЬНИКА

Многоугольники особенно важны в растровой графике как средство задания поверхностей.

Будем называть многоугольник, используемый в качестве окна отсечения, отсекателем, а многоугольник, который отсекается, — отсекаемым.

Алгоритм отсечения многоугольника должен в результате отсечения давать один или несколько замкнутых многоугольников (рис. 0.7.1). При этом могут быть добавлены новые ребра, а имеющиеся или сохранены или разделены или даже отброшены. Существенно, чтобы границы окна, которые не ограничивают видимую часть отсекаемого многоугольника, не входили в состав результата отсечения. Если это не выполняется, то возможна излишняя закраска границ окна (см. рис. 0.7.1б).



а) корректное отсечение

б) некорректное отсечение, так как в состав результата включены лишние фрагменты окна — рёбра Ds-H и P-Gs

Рис. 0.7.1: Отсечение окном PQRS многоугольников ABCDEFGHIJ и KLMN

В принципе эту задачу можно решить с использованием рассмотренных выше алгоритмов отсечения линий, если рассматривать многоугольник просто как набор векторов, а не как сплошные закрашиваемые области. При этом вектора, составляющие многоугольник, просто последовательно отсекаются сторонами окна (рис. 0.7.2).

Если же в результате отсечения должен быть получен замкнутый многоугольник, то формируется вектор, соединяющий последнюю видимую точку с точкой пересечения с окном (рис. 0.7.3а). Проблема возникает при окружении отсекаемым многоугольником угла окна (см. рис. 0.7.3б).

Здесь мы рассмотрим три алгоритма корректно решающие задачу отсечения сплошного многоугольника. Первые два алгоритма быстро работают, но генерируют лишние ребра, как это продемонстрировано на рис. 0.7.1б. Последний алгоритм свободен от указанного недостатка.

В общем, при отсечении многоугольников возникают два типа задач — отображение части изображения попавшей в окно и наоборот, отображение изображения, находящегося вне окна. Все здесь рассматриваемые алгоритмы могут использоваться в обоих случаях.

0.7.1 Алгоритм Сазерленда-Ходгмана

Простой метод решения проблемы охвата отсекаемым многоугольником вершины окна предлагается в алгоритме Сазерленда-Хогдмана [40], когда весь многоугольник последовательно

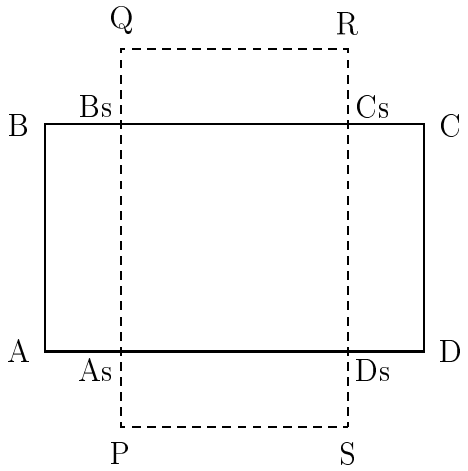


Рис. 0.7.2: Отсечение окном PQRS многоугольника ABCD, рассматриваемого как набор векторов. Генерируется вывод из двух векторов BsCs и DsAs.

отсекается каждой границей окна, как это показано на рис. 0.7.4.

При отсечении ребра, соединяющего очередную пару вершин K и L, возможны 4 случая взаимного расположения (рис. 0.7.5):

а) ребро на внутренней стороне границы,

б) ребро выходит из окна наружу,

в) ребро на внешней стороне границы,

г) ребро входит снаружи в окно.

В случае а) в результат добавляется вершина L. В случае б) в результат заносится S — точка пересечения ребра с границей. В случае в) нет вывода. В случае г) выдаются точка пересечения S и конечная точка ребра L.

Для определения взаимного расположения и направленности используется векторное произведение вектора $\mathbf{P}_1\mathbf{P}_2$, проведенного из начальной в конечную точку текущего ребра окна, на вектор $\mathbf{P}_1\mathbf{S}$ из начальной точки текущего ребра окна в очередную вершину S многоугольника (рис. 0.7.6).

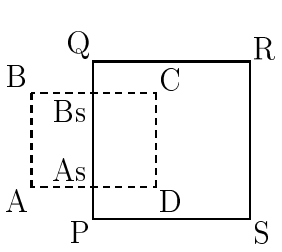
Предложена аппаратная реализация этого алгоритма, состоящая из четырех идентичных ступеней отсечения без промежуточной памяти [28].

В алгоритме Сазерленда-Ходгмана в результат могут заноситься границы окна, даже если они и не ограничивают видимую часть отсеченного многоугольника. Это можно устранить дополнительным анализом, либо используя более сложный алгоритм отсечения.

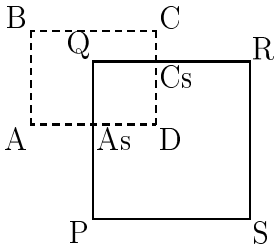
0.7.2 Простой алгоритм отсечения многоугольника

В данном разделе рассматривается простой алгоритм отсечения, который подобно алгоритму Сазерленда-Ходгмана может генерировать лишние стороны для отсеченного многоугольника, проходящие вдоль ребра окна отсечения. Но этот алгоритм несколько более быстрый и использует те же подпрограммы обработки многоугольного окна отсечения, что и алгоритм Кируса-Бека.

Многоугольник отсекается одним ребром выпуклого окна отсечения. В результате такого отсечения формируется новый многоугольник, который затем отсекается следующим ребром и т.д., пока не будет выполнено отсечение последним ребром окна.



а) простой случай. Генерируется вывод из четырех отрезков: BsC, CD, DAs и AsBs, который соединяет последнюю видимую точку As с точкой пересечения Bs.



б) сложный случай. При отсечении векторов сгенерированы отрезки CsD и DAs. Замыкание многоугольника надо выполнять двумя отрезками — AsQ и QCs, а не AsCs.

Рис. 0.7.3: Отсечение сплошного многоугольника окном.

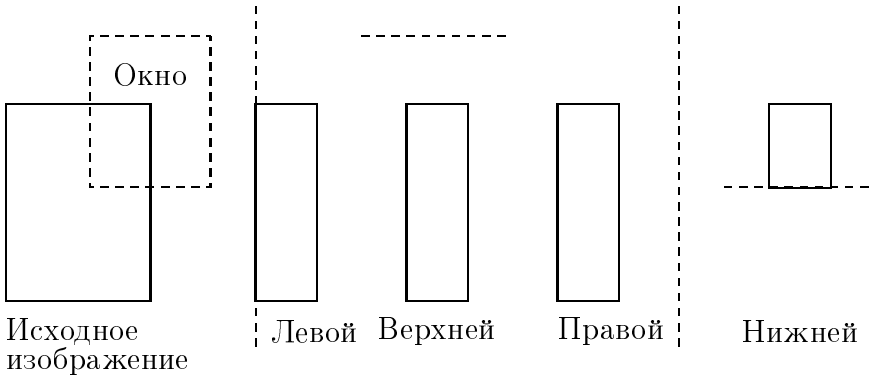


Рис. 0.7.4: Последовательное отсечение многоугольника сторонами окна.

Основная здесь процедура — процедура отсечения отдельным ребром, определяющая взаимное расположение очередной стороны многоугольника и ребра отсекателя и генерирующая соответствующие выходные данные.

Возможны 9 различных случаев расположения ребра окна и отсекаемой стороны, показанных на рис. 0.7.7–0.7.9.

На них V_0 и V_1 — начальная и конечная точки отсекаемой стороны многоугольника, соответственно; $\mathbf{Nr}$ — нормаль к ребру окна отсечения, направленная внутрь окна.

Из этих рисунков очевидны правила генерации выходных данных, зависящие от варианта взаимного расположения:

1) Нет выходных данных.

2) В выходные данные заносится конечная точка.

3) Рассчитывается пересечение и в выходные данные заносятся точка пересечения и конечная точка.

4) Нет выходных данных.

5) В выходные данные заносится конечная точка.

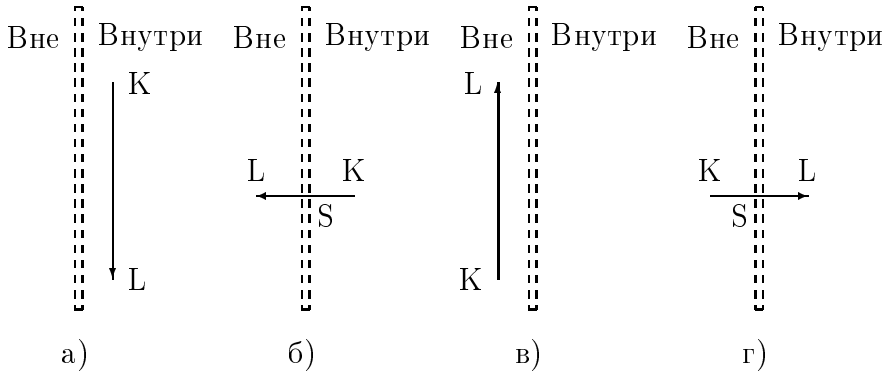
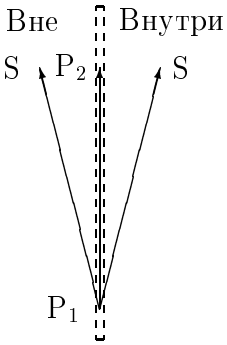


Рис. 0.7.5: Относительные расположения ребра и границы окна.



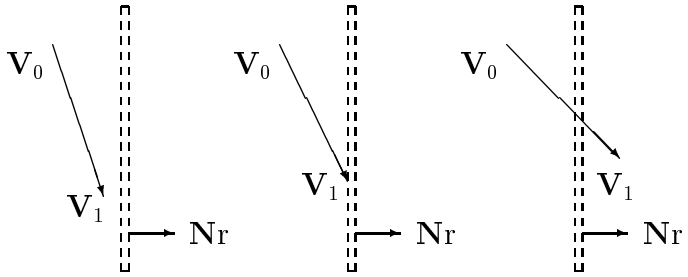
Если $\mathbf{P}_1\mathbf{P}_2 \times \mathbf{P}_1\mathbf{S} < 0$, то поворот от $\mathbf{P}_1\mathbf{P}_2$ $\mathbf{P}_1\mathbf{S}$ по часовой стрелке, т.е. точка S внутри окна.

Если $\mathbf{P}_1\mathbf{P}_2 \times \mathbf{P}_1\mathbf{S} > 0$, то поворот от $\mathbf{P}_1\mathbf{P}_2$ $\mathbf{P}_1\mathbf{S}$ против часовой стрелки, т.е. точка S вне окна.

Рис. 0.7.6: Определение взаимного расположения окна и вершины

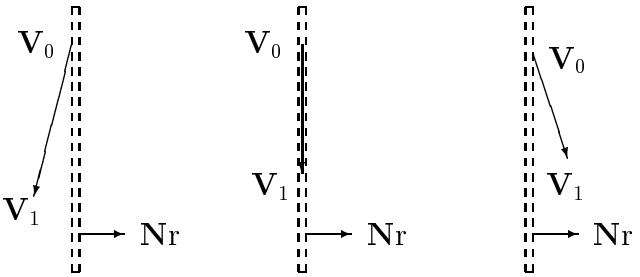
6) В выходные данные заносится конечная точка.
7) Рассчитывается пересечение и в выходные данные заносится только точка пересечения.
8) В выходные данные заносится конечная точка.
9) В выходные данные заносится конечная точка.

Для определения взаимного расположения, подобно алгоритму отсечения Кируса-Бека, используется скалярное произведение Q вектора нормали на вектор, проведенный из начала ребра в анализируемую точку. Пояснение см. на рис. 8.10.



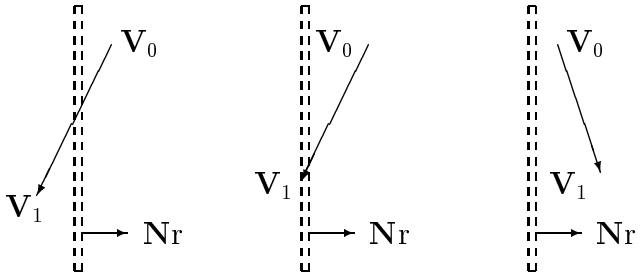
Конечная точка: 1) вне 2) на ребре 3) внутри окна

Рис. 0.7.7: Начальная точка вне ребра окна отсечения.



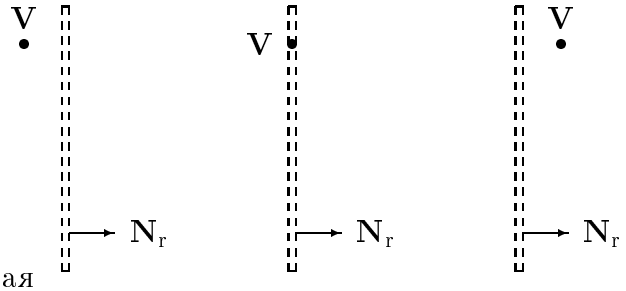
Конечная точка: 4) вне 5) на ребре 6) внутри окна

Рис. 0.7.8: Начальная точка на ребре окна отсечения.



Конечная точка: 7) вне 8) на ребре 9) внутри окна

Рис. 0.7.9: Начальная точка внутри окна отсечения.



Анализируемая точка: 1) вне 2) на ребре 3) внутри окна

Рис. 8.10. Анализ расположения точки относительно ребра окна отсечения.

Таким образом, для определения взаимного расположения начальной $\mathbf{V}_0$ и конечной $\mathbf{V}_1$ точек отсекаемой стороны и ребра отсечения с вектором его начала $\mathbf{R}$, надо вычислить:

$$\mathrm{Qn} = (\mathbf{V}_0 - \mathbf{R}) \cdot \mathbf{N}\mathrm{r}$$

$$\mathrm{Qk} = (\mathbf{V}_1 - \mathbf{R}) \cdot \mathbf{N}\mathrm{r}.$$

Расчет пересечения, если он требуется, производится аналогично алгоритму Кируса-Бека с использованием параметрического представления линии:

$$\mathbf{V}(\mathrm{t}) = \mathbf{V}_0 + (\mathbf{V}_1 - \mathbf{V}_0) \cdot \mathrm{t}.$$

Вначале находится значение параметра t для точки пересечения по формуле (см. описание алгоритма Кируса-Бека):

$$t = -Qn / Pn,$$

где Qn — скалярное произведение вектора нормали к ребру окна на вектор из начала ребра в начальную точку стороны, уже вычисленное при определении расположения начальной точки, а $Pn = (\mathbf{V}_1 - \mathbf{V}_0) \cdot \mathbf{N}r$ — скалярное произведение вектора нормали к ребру окна на вектор из начальной в конечную точки отсекаемой стороны.

Легко выразить это произведение через уже вычисленные величины Qn и Qk:

$$\begin{aligned} Pn &= (\mathbf{V}_1 - \mathbf{V}_0) \cdot \mathbf{N}r \\ &= (\mathbf{V}_1 - \mathbf{V}_0 - \mathbf{R} + \mathbf{R}) \cdot \mathbf{N}r \\ &= (\mathbf{V}_1 - \mathbf{R}) \cdot \mathbf{N}r - (\mathbf{V}_0 - \mathbf{R}) \cdot \mathbf{N}r \\ &= Qk - Qn. \end{aligned}$$

Таким образом, точке пересечения соответствует значение параметра t, равное:

$$t = Qn / (Qn - Qk).$$

Значения координат пересечения находятся из:

$$Xp = X_0 + (X_1 - X_0) \cdot t; \quad Yp = Y_0 + (Y_1 - Y_0) \cdot t.$$

Описанный алгоритм реализован в процедуре V_Plclip, приведенной в Приложении 8.

0.7.3 Алгоритм отсечения многоугольника Вейлера-Азертона

В предыдущих разделах были рассмотрены два алгоритма отсечения многоугольника, последовательно отсекающие произвольный (как выпуклый, так и невыпуклый) многоугольник каждой из сторон выпуклого окна. Зачастую же требуется отсечение по невыпуклому окну. Кроме того оба рассмотренных алгоритма могут генерировать лишние стороны для отсеченного многоугольника, проходящие вдоль ребра окна отсечения. Далее рассматриваемый алгоритм Вейлера-Азертона [?, ?, Род89] свободен от указанных недостатков ценой заметно большей сложности и меньшей скорости работы.

Предполагается, что каждый из многоугольников задан списком вершин, причем таким образом, что при движении по списку вершин в порядке их задания внутренняя область многоугольника находится справа от границы.

В случае пересечения границ и отсекаемого многоугольника и окна возникают точки двух типов:

- входные точки, когда ориентированное ребро отсекаемого многоугольника входит в окно,
- выходные точки, когда ребро отсекаемого многоугольника идет с внутренней на внешнюю стороны окна.

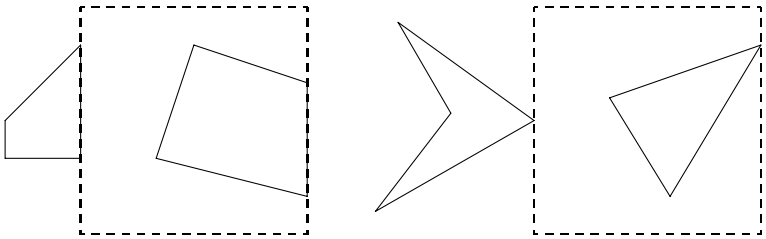
Общая схема алгоритма Вейлера-Азертона для определения части отсекаемого многоугольника, попавшей в окно, следующая:

1. Строятся списки вершин отсекаемого многоугольника и окна.

2. Отыскиваются все точки пересечения. При этом расчете касания не считаются пересечением, т.е. когда вершина или ребро отсекаемого многоугольника инцидентна или совпадает со стороной окна (рис. 0.7.10 и 0.7.11).

3. Списки координат вершин отсекаемого многоугольника и окна дополняются новыми вершинами — координатами точек пересечения. Причем если точка пересечения Pk находится на ребре, соединяющем вершины Vi, Vj, то последовательность точек Vi, Vj превращается в последовательность Vi, Pk, Vj. При этом устанавливаются двухсторонние связи между одноименными точками пересечения в списках вершин отсекаемого многоугольника и окна.
   Входные и выходные точки пересечения образуют отдельные подсписки входных и выходных точек в списках вершин.

4. Определение части обрабатываемого многоугольника, попавшей в окно выполняется следующим образом:
   Если не исчерпан список входных точек пересечения, то выбираем очередную входную точку.
   Двигаемся по вершинам отсекаемого многоугольника пока не обнаружится следующая точка пересечения; все пройденные точки, не включая прервавшую просмотр, заносим в результат; используя двухстороннюю связь точек пересечения, переключаемся на просмотр списка вершин окна.
   Двигаемся по вершинам окна до обнаружения следующей точки пересечения; все пройденные точки, не включая последнюю, прервавшую просмотр, заносим в результат.
   Используя двухстороннюю связь точек пересечения, переключаемся на список вершин обрабатываемого многоугольника.
   Эти действия повторяем пока не будет достигнута исходная вершина — очередная часть отсекаемого многоугольника, попавшая в окно, замкнулась. Переходим на выбор следующей входной точки в списке отсекаемого многоугольника.



Ребро отсекаемого многоугольника совпадает с окном.

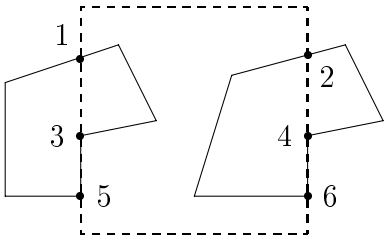
Вершина отсекаемого многоугольника касается окна.

Рис. 0.7.10: Случаи не считающиеся пересечением.

Модификация этого алгоритма для определения части отсекаемого многоугольника, находящейся вне окна, заключается в следующем:

- исходная точка пересечения пересечения берется из списка выходных точек,

- движение по списку вершин окна выполняется в обратном порядке, т.е. так чтобы внутренняя часть отсекателя была слева.



Точки 1–4 считаются пересечением.

Точки 5,6 не считаются пересечением.

Рис. 0.7.11: Частные случаи пересечения.

На рис. 0.7.12 иллюстрируется отсечение многоугольника ABCDEFGHI окном PQRS по алгоритму Вейлера-Азертона.

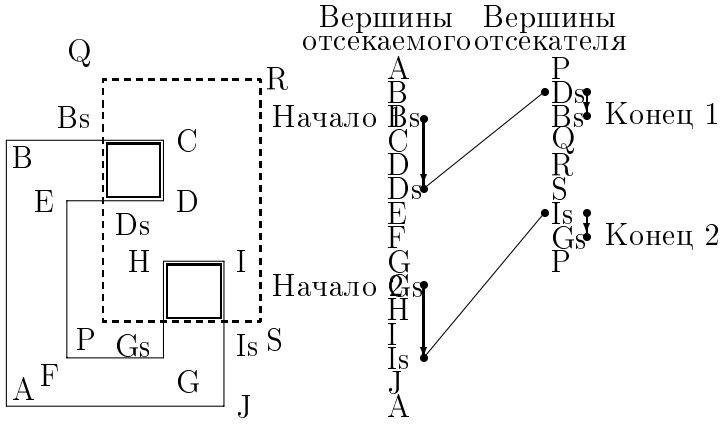


Рис. 0.7.12: Отсечение по алгоритму Вейлера-Азертона.

Начиная с этого алгоритма, при рассмотрении многих дальнейших требуется представления о различных структурах данных и работе с ними. Следующий раздел и посвящен беглому рассмотрению некоторых наиболее важных структур данных.

0.8 СТРУКТУРЫ ДАННЫХ

В данном разделе рассмотрены три основных способа работы с данными:
, и . Данные могут быть организованы в виде:

- элементов данных (data item) — единичная информация, перерабатываемая системой,
- запись (record) — совокупность некоторого числа элементов данных,
- файл (file) — совокупность некоторого числа записей.

В общем случае структура данных образуется посредством упорядочивания записей и связей между ними в файл. В зависимости от требуемых операций данные в файле организуются различным образом.

0.8.1 Последовательный доступ

Зачастую требуется простейшая последовательная обработка записей. Для этого достаточна последовательная организация данных, когда записи запоминаются в файле в последовательности поступления.

Выбор требуемой записи в файле с последовательной организацией возможен только путем его сканирования от начала до требуемой записи.

Удаление и/или вставка не последней записи приводят к большим перемещениям данных (рис. 0.8.1, 0.8.2).

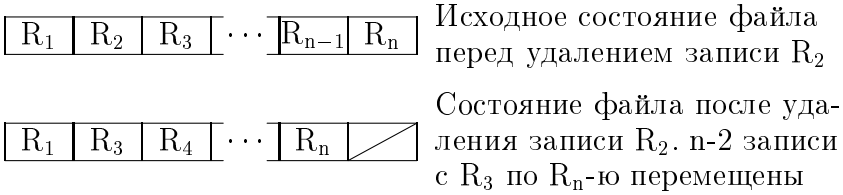


Рис. 0.8.1: Удаление записи из файла с последовательным доступом

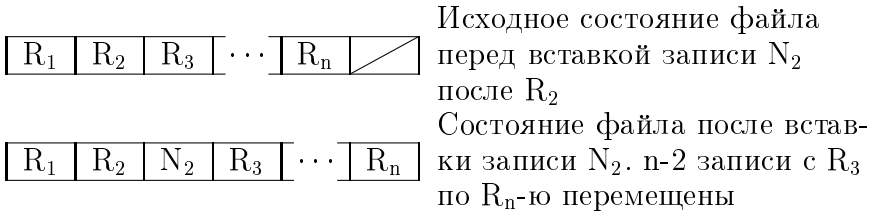


Рис. 0.8.2: Вставка записи в файл с последовательным доступом

Очереди и стеки

Частными, широко используемыми случаями последовательного доступа, являются очереди и стеки. Наиболее широко используются стеки.

Очередь — файл данных с дисциплиной обслуживания первым пришел — первым обслужен (FIFO — First Input First Output) (рис 0.8.3).

Стек (магазин, гнездовая память, память LIFO) — файл данных с дисциплиной обслуживания первым пришел — последним обслужен (LIFO — Last Input First Output).

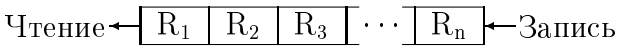


Рис. 0.8.3: Работа с очередью

Состояние стека определяется значением указателя стека, инициируемом при его создании. Для работы с созданным стеком достаточны две операции:

- PUSH — помещение записи в стек,
- POP — получение записи из стека.

Эти операции заносят/читают данные и модифицируют значение указателя стека. Стек может быть организован двумя способами, отличающимися правилами продвижения указателя стека:

1. Указатель стека указывает на последнюю занесенную запись.
   До занесения указатель стека уменьшается на длину записи (рис. 0.8.4).
   После чтения указатель стека увеличивается на длину записи (рис. 0.8.5).

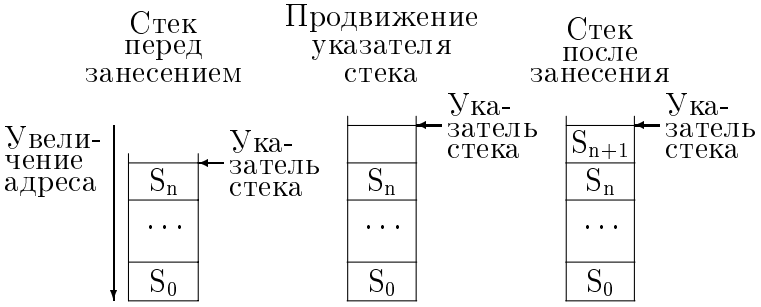


Рис. 0.8.4: Запись данных в стек типа 1

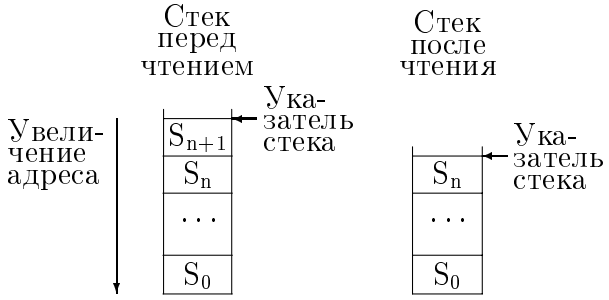


Рис. 0.8.5: Чтение данных из стека типа 1

2. Указатель стека указывает на свободное место в стеке.
   После занесения указатель стека увеличивается на длину записи (рис. 0.8.6).
   Перед чтением указатель стека уменьшается на длину записи. (рис. 0.8.7).

Наиболее часто используется первый способ организации стека.

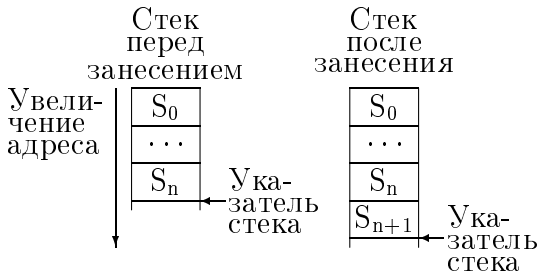


Рис. 0.8.6: Запись данных в стек типа 2

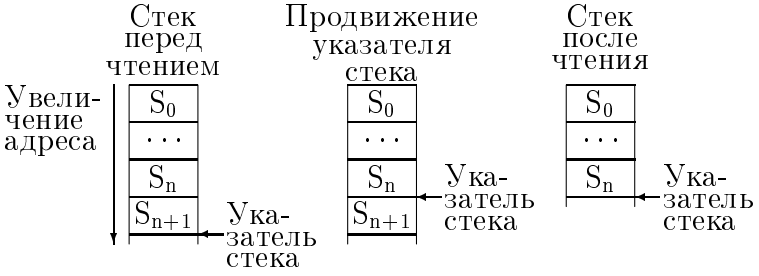


Рис. 0.8.7: Чтение данных из стека типа 2

0.8.2 Непосредственный доступ

При необходимости непосредственного доступа (random access) устанавливается связь между адресом запоминания и ключом, по которому ищется запись.

Простой непосредственный доступ

В простейшем случае ключ поиска — просто номер записи. Чаще ключ — некоторая совокупность данных из записи. По значению ключа либо непосредственно вычисляется адрес записи (например, адрес расположения физического блока фиксированной длины на диске вычисляется по его номеру), либо ключ используется для доступа к справочнику, в котором отыскивается адрес (например, адрес начала расположения i-го файла задается i-й записью в директории). Метод очень быстрый — мы немедленно получаем доступ к записи. Но он может быть очень расточительным по использованию памяти, когда из полного возможного набора в N ключей, по которым вырабатываются адреса в диапазоне от 0 до N-1, фактически будет иметься только K $\ll$ N ключей, т.е. из всего диапазона адресов 0-(N-1) будет использоваться только K адресов, раскиданных по всему объему от 0 до N-1.

Непосредственный доступ с использованием хеш-адресации

В таких случаях значение ключа используется для вычисления функции расстановки (hash code), определяющей адрес расположения данных. Причем функция расстановки подбирается такой, чтобы для K ключей из полного допустимого набора в N ключей, генерируемое количество адресов L было возможно более близко к K. Этот подход обычно используется при поиске информации об объекте по его имени, например, для поиска информации об объекте по его

наименованию, для поиска информации в базе данных о сотруднике по его фамилии, для работы с таблицей идентификаторов в трансляторах и т.д. В этом случае функция расстановки вычисляется из символов ключа. Идеальная функция расстановки должна вычислять уникальный адрес для каждого из фактических ключей. Реальные же функции расстановки могут для разных ключей давать один и тот же адрес. Рассмотрим частный пример (табл. 9.1) занесения информации в таблицу из 10 элементов (L равно 10). В качестве ключа используем наименование объекта. Для вычисления хеш-адреса Hn в диапазоне 0-9 суммируем коды символов ключа, из которых предварительно вычитаем 65. В качестве хеш-адреса берем остаток от деления полученной суммы на 10. (На самом деле число 10 не годится, а выбрано для наглядности).

Пример вычисления хеш-адреса. Таблица 9.1

Ключ	Коды символов ключа							Хеш-адрес
GALLERY	71	65	76	76	69	82	89	3
HOUSE	72	79	85	83	69			3
MIRROR	77	73	82	82	79	82		5
RING	82	73	78	71				4
SCENE	83	67	69	78	69			1

Из табл. 9.1. видно, что для двух разных ключей GALLERY и HOUSE вычислен одинаковый хеш-адрес, равный 3. Предложен ряд способов разрешения таких коллизий. Рассмотрим один из них, называемый рехешированием.

Перед занесением очередной строки в таблицу проверяем занят ли элемент. Если элемент таблицы свободен, то выполняем занесение. Если же элемент таблицы уже занят, то сравниваем ключи So уже занесенного и Sn заносимого элементов. Если они совпали, то выполняем какую-либо процедуру реагирования на одинаковые ключи и завершаем занесение. Например, для базы данных по кадрам выдаем диагностику о том что информация о данном сотруднике уже есть, для таблицы идентификаторов формируем сообщение о повторном описании и т.п. Если же ключи So и Sn не совпали, то вычисляется новый адрес в таблице по формуле:

$$H_i = H_n + P_i \quad L, \tag{0.8.1}$$

где H_n — исходный хеш-адрес, P_i — некоторое число, L — длина таблицы.

Если этот элемент также оказался занятым, то задается новое значение P_i и по формуле (0.8.1) вычисляется следующий адрес т.д., пока не будет найден некоторый элемент, который либо пуст, либо содержит ключ Sn, либо не будет получен исходный хеш-адрес. В последнем случае работа прекращается из-за переполнения таблицы. Используются несколько основных способов рехеширования [Гри75, Хол78]:

Линейное рехеширование. В этом, наиболее распространенном и простом случае P_1 равно 1, P_2 равно 2 и т.д.
Число сравнений E ≈ (1 - V/2)(1-V), где V — коэффициент заполненности таблицы.
10% — E = 1.06 сравнений,
50% — E = 1.50 сравнений,
90% — E = 5.50 сравнений.

Случайное рехеширование. При этом способе P_j — псевдослучайное число.
Способ хорош при $L = 2^m$ где m — целое.
Число сравнений E ≈ - (1/V)log(1-V).
10% — E = 1.05 сравнений,

50% — E = 1.39 сравнений,
90% — E = 2.56 сравнений.

Рехеширование сложением с выбором. При этом способе $P_i = i \times H_n(i = 1 \ldots L - 1)$,.
Способ хорош когда L — простое число.

Квадратичное рехеширование. При этом способе $P_i = A \times (i^2) + B \times i + C$.
Здесь A, B, C — произвольные числа, выбор которых определяется эффективностью вычисления формулы на конкретной машине.
Время вычислений хеш-адреса меньше чем при случайном рехешировании.
Если L — простое число, то $E \leq L/2$.

Третья форма организации данных — списки, когда логическая и физическая организация данных разделены и для доступа к данным используются указатели на данные.

Далее будем рассматривать структуры данных с использованием списков указателей.

0.8.3 Линейные списки

Линейный список — множество элементов с помощью которых свойства структуры задаются посредством линейного (одномерного) относительного расположения элементов. При этом k-й указатель непосредственно находится после (k-1)-го.

С таким списком возможны следующие операции:

1. Определение числа элементов в списке.
2. Доступ к k-му элементу для использования и/или изменения его содержимого.
3. Вставка нового элемента.
4. Удаление элемента.
5. Комбинация нескольких списков в один общий список.
6. Разделение списка на части.
7. Поиск.
8. Сортировка.

Пример линейного списка показан на рис. 0.8.8.

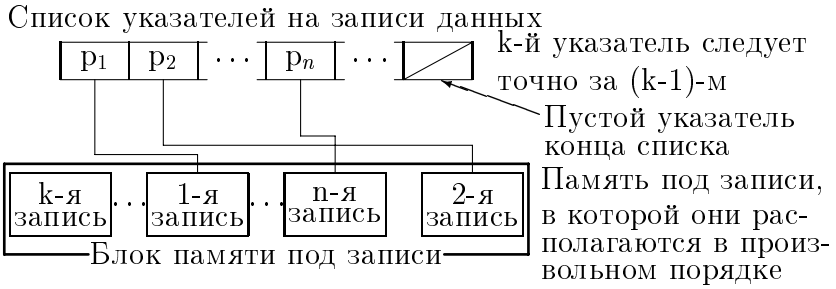


Рис. 0.8.8: Линейный список

0.8.4 Комбинированные списки

Очевидно, что процесс удаления может приводить к большим перемещениям данных. Пусть, например, надо удалить запись номер 1 (см. рис. 0.8.8). Это потребует удаления указателя p_1 и переписи всех далее расположенных указателей на освобождаемое место списка. Аналогичные проблемы возникают и при вставке в требуемое место.

Поэтому естественным шагом является построение так называемых комбинированных списков, содержащих кроме указателей на записи данных также и дополнительные указатели, указывающие на на следующий элемент списка указателей (рис. 0.8.9).

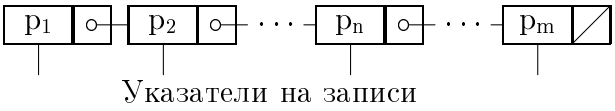


Рис. 0.8.9: Комбинированный список

Достоинства таких списков:

- много проще вставка в требуемое место, например, вставка нового элемента между (n-1)-м и n-м сводится к перестройке двух указателей (рис. 0.8.10)

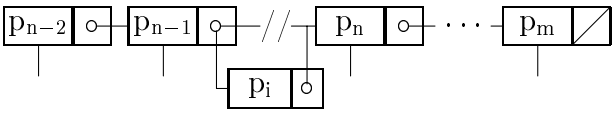


Рис. 0.8.10: Вставка элемента p_i в комбинированный список

- много проще удаление (рис. 0.8.11), которое сводится к перестройке трех указателей — одного указателя для собственно удаления и перестройке двух указателей для включения удаляемого элемента в список свободной памяти.

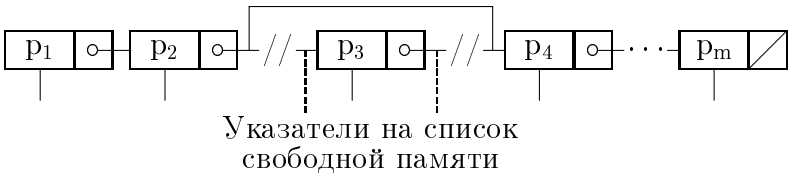


Рис. 0.8.11: Удаление элемента p_3 из комбинированного списка

- разделение и соединение списков упрощается,
- возможно формирование сложных структур данных,
- можно надстраивать переменное число списков различных длин,
- любой из элементов списка может быть заголовком некоторого другого списка, когда указатель на данные указывает на некоторый другой список,
- за счет использования нескольких указателей возможно формирование подсписков.

Недостатки:

- увеличивается потребность в памяти из-за дополнительных указателей,

- обычно требуемая последовательная обработка замедляется,
- выбор нужного элемента списка требует последовательного прохода по всем предшествующим элементам списка, начиная с начального.

0.8.5 Циклические списки

Последнего недостатка лишены списки с указателями не только вперед, на следующий элемент, но и назад, на предшествующий элемент. Эти списки чаще всего используются в виде циклических, когда первый элемент указывает на последний, как предшествующий ему. В свою очередь последний элемент списка как на последующий указывает на начальный элемент. Доступ к началу списка обеспечивает специальный элемент — заголовок списка, который может сопровождаться данными, характеризующими список в целом (рис. 0.8.12). Например, если список представляет собой описание какой-либо фигуры (тела), то дополнительными данными могут быть идентификатор и тип фигуры или иная семантическая информация.

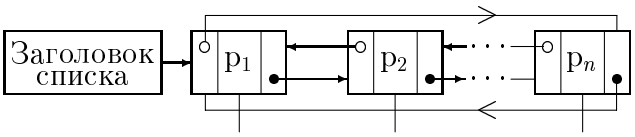


Рис. 0.8.12: Циклический список

Ясно, что удаление или вставка элемента в таком списке также упрощаются.

0.9 ГЕОМЕТРИЧЕСКОЕ МОДЕЛИРОВАНИЕ

Во многих приложениях машинной графики возникает потребность в представлении трехмерных тел (вычислительный эксперимент, автоматизация проектирования, роботизация, вычислительная томография, тренажеры, видеографика и т.д.).

Можно выделить две основные задачи, связанные с представлением трехмерных тел, — построение модели уже существующего объекта и синтез модели заранее не существовавшего объекта.

При решении первой задачи в общем случае может потребоваться задание бесконечного количества координат точек. Чаще же всего объект с той или иной точностью аппроксимируют некоторым конечным набором элементов, например, поверхностей, тел и т.п.

При решении второй задачи, выполняемой чаще всего в интерактивном режиме, основное требование к средствам формирования и представления модели — удобство манипулирования.

Используются три основных типа 3D моделей:

- каркасное представление, когда тело описывается набором ребер,
- поверхностное, когда тело описывается набором ограничивающих его поверхностей,
- модель сплошных тел, когда тело формируется из отдельных базовых геометрических и, возможно, конструктивно — технологических объемных элементов с помощью операций объединения, пересечения, вычитания и преобразований.

Важно отметить, что 3D системы существенно ориентируются на область приложений так как многие характерные для них задачи, выполняемые программным путем, стоят очень дорого и сильно зависят от выбора возможных моделей. Типичными такими задачами, в частности, являются получение сечений и удаление невидимых частей изображения. Обычно имеется много вариантов реализации различных моделей в бóльшей или меньшей степени эффективных в зависимости от различных областей приложений и решаемых задач. Поэтому в 3D системах стремятся использовать многообразие моделей и поддерживать средства перехода от одной модели к другой.

Другим важным обстоятельством является то, что для современных систем характерно стремление моделировать логику работы, принятую пользователем. Это требует наличия средств перехода от модели, удобной для пользователя, к модели удобной для визуализации (модели тел в виде граней).

0.9.1 Элементы моделей

При формировании 3D модели используются:

- двумерные элементы (точки, прямые, отрезки прямых, окружности и их дуги, различные плоские кривые и контуры),
- поверхности (плоскости, поверхности, представленные семейством образующих, поверхности вращения, криволинейные поверхности),
- объемные элементы (параллелепипеды, призмы, пирамиды, конусы, произвольные многогранники и т.п.).

Из этих элементов с помощью различных операций формируется внутреннее представление модели.

0.9.2 Методы построения моделей

Используются два основных способа формирования геометрических элементов моделей — это построение по заданным отношениям (ограничениям) и построение с использованием преобразований.

Построение с использованием отношений

Построение с использованием отношений заключается в том, что задаются:

- элемент подлежащий построению,
- список отношений и элементы к которым относятся отношения.

Например, построение прямой, проходящей через точку пересечения двух других прямых и касательную к окружности.

Используется два способа реализации построения по отношениям — общий и частный.

При общем способе реализации построение по заданным отношениям можно представить в виде двухшаговой процедуры:

- на основе заданных типов отношений, элементов и параметров строится система алгебраических уравнений,
- решается построенная система уравнений.

Очевидное достоинство такого способа — простота расширения системы — для введения нового отношения достаточно просто написать соответствующие уравнения.

Основные проблемы такого способа заключаются в следующем:

- построенная система уравнений может иметь несколько решений, поэтому требуется выбрать одно из них, например, в диалоговом режиме,
- система уравнений может оказаться нелинейной, решаемой приближенными методами, что может потребовать диалога для выбора метода(ов) приближенного решения.

В связи с отмеченными проблемами общий подход реализован только в наиболее современных системах и при достаточно высоком уровне разработчиков в области вычислительной математики [Вел94, ?].

Большинство же систем реализует частный подход, первым приходящий в голову и заключающийся в том, что для каждой триады, включающей строящийся элемент, тип отношения и иные элементы, затрагиваемые отношением, пишется отдельная подпрограмма (например построение прямой, касательной к окружности в заданной точке). Требуемое построение осуществляется выбором из меню и тем или иным вводом требуемых данных [Баз94, ?, ?].

Преимущества такого подхода ясны — проще писать систему. Не менее очевидны и недостатки, когда пользователю требуется использовать сильно разветвленные меню и/или запоминать мало вразумительные сокращения или пиктограммы, так как обычно число требуемых вариантов построения исчисляется сотнями. Расширение системы, реализуемое добавлением новой подпрограммы, требует ее перепроектирования, по крайней мере в части обеспечения доступа пользователя к новым возможностям. В некотором смысле предельный пример этого подхода — система AutoCAD фирмы Autodesk. Авторы даже гордятся сложностью системы: “AutoCAD предоставляет эту крайне сложную технологию” (Предисловие к Справочному руководству AutoCAD версии 2.5).

Понятно, что перспективы за общим подходом с разумным использованием частных решений. Вместе с тем устаревшие системы типа AutoCad скорее всего также будут продолжать

использоваться в силу распространенности, сложившегося круга обученных пользователей и т.п.

Построение с использованием преобразований

Построение нового объекта с использованием преобразований заключается в следующем:

- задается преобразуемый объект,
- задается преобразование (это может быть обычное аффинное преобразование, определяемое матрицей, или некоторое деформирующее преобразование, например, замена одного отрезка контура ломаной),
- выполнение преобразования; в случае аффинного преобразования для векторов всех характерных точек преобразуемого объекта выполняется умножение на матрицу; для углов вначале переходят к точкам и затем выполняют преобразование.

Построение кривых

Важное значение при формировании как 2D, так и 3D моделей имеет построение элементарных кривых. Кривые строятся, в основном, следующими способами:

- той или иной интерполяцией по точкам,
- вычислением конических сечений,
- расчетом пересечения поверхностей,
- выполнением преобразования некоторой кривой,
- формированием замкнутых или разомкнутых контуров из отдельных сегментов, например, отрезков прямых, дуг конических сечений или произвольных кривых.

В качестве последних обычно используются параметрические кубические кривые, так как это наименьшая степень при которой обеспечиваются:

- непрерывность значения первой (второй) производной в точках сшивки сегментов кривых,

- возможность задания неплоских кривых.

Параметрическое представление кривых выбирается по целому ряду причин, в том числе потому, что зачастую объекты могут иметь вертикальные касательные. При этом аппроксимация кривой y = f(x) аналитическими функциями была бы невозможной. Кроме того кривые, которые надо представлять, могут быть неплоскими и незамкнутыми. Наконец, параметрическое представление обеспечивает независимость представления от выбора системы координат и соответствует процессу их отображения на устройствах: позиция естественным образом определяется как две функции времени x(t) и y(t).

В общем виде параметрические кубические кривые можно представить в форме:

$$\begin{array}{rcccccccc} x(t) = & A_{11}\,t^3 & + & A_{12}\,t^2 & + & A_{13}\,t & + & A_{14} \\ y(t) = & A_{21}\,t^3 & + & A_{22}\,t^2 & + & A_{23}\,t & + & A_{24} \\ z(t) = & A_{31}\,t^3 & + & A_{32}\,t^2 & + & A_{33}\,t & + & A_{34} \end{array} \qquad (0.9.1)$$

где параметр t можно считать изменяющимся в диапазоне от 0 до 1, так как интересуют конечные отрезки.

Существует много методов описания параметрических кубических кривых. К наиболее применяемым относятся:

- метод Безье, широко используемый в интерактивных приложениях; в нем задаются положения конечных точек кривой, а значения первой производной задаются неявно с помощью двух других точек, обычно не лежащих на кривой;
- метод B-сплайнов, при котором конечные точки не лежат на кривой и на концах сегментов обеспечивается непрерывность первой и второй производных.

В форме Безье кривая в общем случае задается в виде полинома Бернштейна:

$$\mathrm{P(t)} = \sum_{\mathrm{i=0}}^{\mathrm{n}} \mathrm{Cm_i}\ \mathrm{t^i}\ (1-\mathrm{t})^{\mathrm{m-1}}\ \mathrm{P_i}$$

где $\mathrm{P_i}$ — значения координат в вершинах ломаной, используемой в качестве управляющей ломаной для кривой, t — параметр,

$$\mathrm{Cm_i} = \frac{\mathrm{m!}}{\mathrm{i!\ (m-i)!}}$$

При этом крайние точки управляющей ломаной и кривой совпадают, а наклоны первого и последнего звеньев ломаной совпадают с наклоном кривой в соответствующих точках.

Предложены различные быстрые схемы для вычисления кривой Безье.

В более общей форме B-сплайнов кривая в общем случае задается соотношением:

$$\mathrm{P(t)} = \sum_{\mathrm{i=0}}^{\mathrm{n}} \mathrm{P_i}\ \mathrm{N_{im}(t)}$$

где $\mathrm{P_i}$ — значения координат в вершинах ломаной, используемой в качестве управляющей ломаной для кривой, t — параметр, $\mathrm{N_{im}}$ — весовые функции, определяемые рекуррентным соотношением:

$$N_{i,1} = \begin{cases} 1, & \text{если } x_i \ \le\ t\ \le\ x_{i+1} \\ 0, & \end{cases}$$

$$N_{i,k}(t) = \frac{(t-x_i)\ N_{i,k-1}(t)}{x_{i+k-1}-x_i} + \frac{(x_{i+k}-t)\ N_{i+1,k-1}(t)}{x_{i+k}-x_{i+1}}.$$

Используются и многие другие методы, например, метод Эрмита, при котором задаются положения конечных точек кривой и значения первой производной в них.

Общее в упомянутых подходах состоит в том, что искомая кривая строится с использованием набора управляющих точек.

Построение поверхностей

Основные способы построения поверхностей:

- интерполяцией по точкам,
- перемещением образующей кривой по заданной траектории (кинематический метод),

- деформацией исходной поверхности,
- построением поверхности эквидистантной к исходной,
- кинематический принцип,
- операции добавления/удаления в структуре,
- теоретико-множественные (булевские) операции.

Широко используется бикубические параметрические куски, с помощью которых сложная криволинейная поверхность аппроксимируется набором отдельных кусков с обеспечением непрерывности значения функции и первой (второй) производной при переходе от одного куска к другому. В общем случае представление бикубического параметрического куска имеет вид (приведена формула для x-координаты, для других координат формула аналогична):

$$\begin{aligned} x(s,t) = {} & A_{11}\, s^3\, t^3 + A_{12}\, s^3\, t^2 + A_{13}\, s^3\, t + A_{14}\, s^3 + \\ & A_{21}\, s^2\, t^3 + A_{22}\, s^2\, t^2 + A_{23}\, s^2\, t + A_{24}\, s^2 + \\ & A_{31}\, s\, t^3 + A_{32}\, s\, t^2 + A_{33}\, s\, t + A_{34}\, s + \\ & A_{41}\, t^3 + A_{42}\, t^2 + A_{43}\, t + A_{44}. \end{aligned}$$

Аналогично случаю с параметрическими кубическими кривыми, наиболее применимыми являются:

- форма Безье,
- форма B-сплайнов,
- форма Эрмита.

0.9.3 Типы моделей

Как уже отмечалось, можно выделить два основных типа представлений 3D моделей:

- , когда в модели хранятся границы объекта, например, вершины, ребра, грани,
- в виде , когда хранятся базовые объекты (призма, пирамида, цилиндр, конус и т.п.) из которых формировалось тело и использованные при этом операции; в узле дерева сохраняется операция формирования, а ветви представляют объекты.

Предельным случаем граничной модели является модель, использующая всех точек занимаемого ею пространства. В частности, тело может быть аппроксимировано набором "склеенных" друг с другом параллелепипедов, что может быть удобно для некоторых вычислений (веса, объемы, расчеты методом конечных элементов и т.д.).

Часто используются гибридные модели, в которых в различной мере смешиваются эти два основных типа представления. В частности, в граничной модели может сохраняться информация о способе построения, например, информация о контуре и траектории его перемещения для формирования заданной поверхности (это т.н. модели). В моделях в виде дерева построения в качестве элементарных могут использоваться не только базовые объекты, но также и сплошные тела, заданные с помощью границ.

В общем случае нельзя утверждать, что одна модель во всем лучше другой. Так, например, граничная модель удобна для выполнения операций визуализации (удаление невидимых частей, закраска и т.п.), с другой стороны модель в виде дерева построения естественным образом может обеспечить параметризацию объекта, т.е. модификацию объекта изменением тех или иных отдельных параметров, вплоть до убирания каких-либо составных частей, но не удобна для визуализации, так как требует перевычисления объекта по дереву построения. Поэтому необходимы средства взаимного преобразования моделей. Понятно, что из более общей можно сформировать более простую, обратное преобразование далеко не всегда возможно или целесообразно, что и иллюстрируется сплошными и штриховыми линиями на рис. 0.9.1.

Из рис. 0.9.1 видно особое место граничной модели, преобразование в которую возможно из

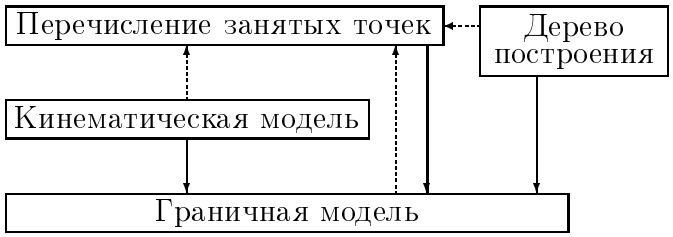


Рис. 0.9.1: Преобразования моделей представления

любых других[1]. Учитывая это, а также и то, что эта модель наиболее удобна для визуализации дальнейшее рассмотрение будет, в основном, относиться к этой модели.

Используются две основных разновидности способов представления поверхностей тела:

- представление в виде набора вершин, ребер и плоских многоугольников (полигональных сеток),
- представление с использованием параметрических бикубических площадок (кусков).

Полигональные сетки используются как для представления плоских поверхностей, так и для аппроксимации криволинейных, в том числе и параметрических бикубических площадок, поэтому далее в основном подразумевается представление поверхности в виде плоских многоугольников.

0.9.4 Полигональные сетки

Как отмечалось, полигональная сетка представляет собой набор вершин, ребер и плоских многоугольников. Вершины соединяются ребрами. Многоугольники рассматриваются либо как последовательность вершин или ребер. Можно предложить много способов внутреннего представления полигональных сеток.

На рис. 0.9.2 изображен простой пример полигональной сетки из четырех многоугольников с девятью вершинами и двенадцатью ребрами. На рис. 0.9.3–0.9.5 рассмотрены несколько различных представлений и приведены соображения по их эффективности и удобству манипулирования.

0.9.5 Внутреннее представление моделей

- объемная модель,
- полиэдральная модель,
- упрощенные модели,
- сосуществование моделей.

Процедурная модель и представление в данных.

[1] Переход к модели с перечислением занятых точек также возможен из любой другой, но при решении проблем точности аппроксимации.

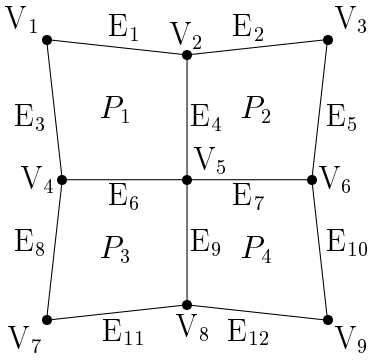


Рис. 0.9.2: Пример полигональной сетки. P_i — многоугольники, V_j — вершины, E_k — ребра.

P_1	P_2	P_3	P_4
V_1	V_2	V_4	V_5
V_2	V_3	V_5	V_6
V_5	V_6	V_8	V_9
V_4	V_5	V_7	V_8

Рис. 0.9.3: Представление полигональной сетки с явным заданием многоугольников. Компактно для одного многоугольника, но сильно избыточно для набора, так как не существует общего описания общих вершин и ребер.

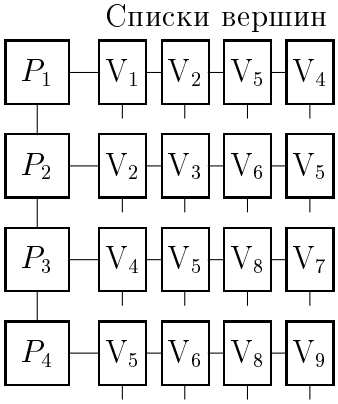


Рис. 0.9.4: Представление полигональной сетки с указателями на списки вершин. Элементы списка указателей на вершины для каждого многоугольника ссылаются на соответствующие координатные данные для вершин. Данное представление компактнее предыдущего, но трудно найти многоугольники с общими ребрами.

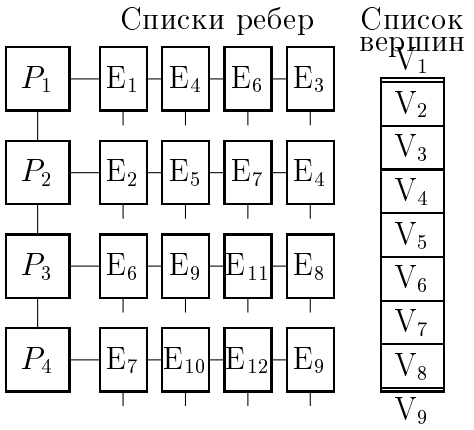


Рис. 0.9.5: Представление полигональной сетки в виде списка ребер. Элементы списка ребер содержат указатели на вершины в списке вершин, образующие данное ребро. Для обеспечения поиска всех вершин, образующих данный многоугольник, необходимо иметь обратные указатели от вершины на одно из инцидентных к ней ребер.

0.10 УДАЛЕНИЕ СКРЫТЫХ ЛИНИЙ И ПОВЕРХНОСТЕЙ

0.10.1 Классификация методов удаления невидимых частей

Методы удаления невидимых частей сцены можно классифицировать:

1. По выбору удаляемых частей:
   удаление невидимых линий, ребер, поверхностей, объемов.
2. По порядку обработки элементов сцены:
   удаление в произвольном порядке и в порядке, определяемом процессом визуализации.
3. По системе координат:
   - алгоритмы работающие в пространстве объектов, когда каждая из N граней объекта сравнивается с остальными N-1 гранями (объем вычислений растет как N^2),
   - алгоритмы работающие в пространстве изображения, когда для каждого пиксела изображения определяется какая из N граней объекта видна (при разрешении экрана M×M объем вычислений растет как $M^2 \times N$).

0.10.2 Алгоритмы удаления линий

Применение — векторные устройства. Могут применяться и в растровых для ускорения процесса визуализации, но при этом не используется основное ценное качество растрового дисплея — возможность закраски поверхностей.

Наиболее известный ранний алгоритм — алгоритм Робертса (1963 г.). Работает с только выпуклыми телами в пространстве объектов. Каждый объект сцены представляется многогранным телом, полученным в результате пересечения плоскостей. Т.е. тело описывается списком граней, состоящих из ребер, которые в свою очередь образованы вершинами.

Вначале из описания каждого тела удаляются нелицевые плоскости, экранированные самим телом. Затем каждое из ребер сравнивается с каждым телом для определения видимости или невидимости. Т.е. объем вычислений растет как квадрат числа объектов в сцене. Наконец вычисляются новые ребра, полученные при протыкании телами друг друга.

0.10.3 Алгоритм удаления поверхностей с Z-буфером

Алгоритм предложен Эдом Кэтмулом и представляет собой обобщение буфера кадра. Обычный буфер кадра хранит коды цвета для каждого пиксела в пространстве изображения. Идея алгоритма состоит в том, чтобы для каждого пиксела дополнительно хранить еще и координату Z или глубину. При занесении очередного пиксела в буфер кадра значение его Z-координаты сравнивается с Z-координатой пиксела, который уже находится в буфере. Если Z-координата нового пиксела больше, чем координата старого, т.е. он ближе к наблюдателю, то атрибуты нового пиксела и его Z-координата заносятся в буфер, если нет, то ни чего не делается.

Этот алгоритм наиболее простой из всех алгоритмов удаления невидимых поверхностей, но требует большого объема памяти. Данные о глубине для реалистичности изображения обычно достаточно иметь с разрядностью порядка 20 бит. В этом случае при изображении нормального телевизионного размера в 768×576 пикселов для хранения Z-координат необходим объем

памяти порядка 1 Мбайта. Суммарный объем памяти при 3 байтах для значений RGB составит более 2.3 Мбайта.

Время работы алгоритма не зависит от сложности сцены. Многоугольники, составляющие сцену, могут обрабатываться в произвольном порядке. Для сокращения затрат времени нелицевые многоугольники могут быть удалены. По сути дела алгоритм с Z-буфером — некоторая модификация уже рассмотренного алгоритма заливки многоугольника. Если используется построчный алгоритм заливки, то легко сделать пошаговое вычисление Z-координаты очередного пиксела, дополнительно храня Z-координаты его вершин и вычисляя приращение dz Z-координаты при перемещении вдоль X на dx, равное 1. Если известно уравнение плоскости, в которой лежит обрабатываемый многоугольник, то можно обойтись без хранения Z-координат вершин. Пусть уравнение плоскости имеет вид:

$$A \cdot x + B \cdot y + C \cdot z + D = 0.$$

Тогда при C не равном нулю

$$z = -(A \cdot x + B \cdot y + D)/C$$

Найдем приращение Z-координаты пиксела при шаге по X на dx, помня, что Y очередной обрабатываемой строки — константа.

$$dz = -(A \cdot (x + dx) + D)/C + (A \cdot x + D)/C = -A \cdot dx/C$$

но $dx = 1$, поэтому

$$dz = -A/C.$$

Основной недостаток алгоритма с Z-буфером — дополнительные затраты памяти. Для их уменьшения можно разбивать изображение на несколько прямоугольников или полос. В пределе можно использовать Z-буфер в виде одной строки. Понятно, что это приведет к увеличению времени, так как каждый прямоугольник будет обрабатываться столько раз, на сколько областей разбито пространство изображения. Уменьшение затрат времени в этом случае может быть обеспечено предварительной сортировкой многоугольников на плоскости.

Другие недостатки алгоритма с Z-буфером заключаются в том, что так как пикселы в буфер заносятся в произвольном порядке, то возникают трудности с реализацией эффектов прозрачности или просвечивания и устранением лестничного эффекта с использованием предфильтрации, когда каждый пиксел экрана трактуется как точка конечного размера и его атрибуты устанавливаются в зависимости от того какая часть пиксела изображения попадает в пиксел экрана. Но другой подход к устранению лестничного эффекта, основанный на постфильтрации — усреднении значений пиксела с использованием изображения с большим разрешением реализуется сравнительно просто за счет увеличения расхода памяти (и времени). В этом случае используются два метода. Первый состоит в том, что увеличивается разрешение только кадрового буфера, хранящего атрибуты пикселов, а разрешение Z-буфера делается совпадающим с размерами пространства изображения. Глубина изображения вычисляется только для центра группы усредняемых пикселов. Это метод неприменим, когда расстояние до наблюдателя имитируется изменением интенсивности пикселов. Во втором методе и кадровый и Z буфера имеют увеличенное разрешение и усредняются атрибуты пиксела, так и его глубина.

Общая схема алгоритма с Z-буфером:

- Инициализировать кадровый и Z-буфера. Кадровый буфер закрашивается фоном. Z-буфер закрашивается минимальным значением Z.
- Выполнить преобразование каждого многоугольника сцены в растровую форму. При этом для каждого пиксела вычисляется его глубина z. Если вычисленная глубина больше, чем глубина, уже имеющаяся в Z-буфере, то занести в буфера атрибуты пиксела и его глубину, иначе никаких занесений не выполнять.
- Выполнить, если это было предусмотрено, усреднение изображения с понижением разрешения.

0.10.4 Построчный алгоритм с Z-буфером

Рассмотрим теперь алгоритм с Z-буфером размером в одну строку, который представляет собой обобщение алгоритма построчной заливки многоугольника, представленный в процедурах V_FP0 и V_FP1 в приложениях. Модификация должна учесть то, что для каждой строки сканирования теперь может обрабатываться не один многоугольник.

Общая схема такого алгоритма следующая:

1. Подготовка данных.
   Для каждого многоугольника определить максимальную Y-координату.
   Занести многоугольник в группу многоугольников, соответствующую данной Y-координате.

2. Собственно заливка.

0.10.5 Алгоритм разбиения области Варнока

Алгоритм работает в пространстве изображения и анализирует область на экране дисплея (окно) на наличие в них видимых элементов. Если в окне нет изображения, то оно просто закрашивается фоном. Если же в окне имеется элемент, то проверяется достаточно ли он прост для визуализации. Если объект сложный, то окно разбивается на более мелкие, для каждого из которых выполняется тест на отсутствие и/или простоту изображения. Рекурсивный процесс разбиения может продолжаться до тех пор пока не будет достигнут предел разрешения экрана.

Можно выделить 4 случая взаимного расположения окна и многоугольника (рис. 0.10.1):

- многоугольник целиком вне окна,
- многоугольник целиком внутри окна,
- многоугольник пересекает окно,
- многоугольник охватывает окно.

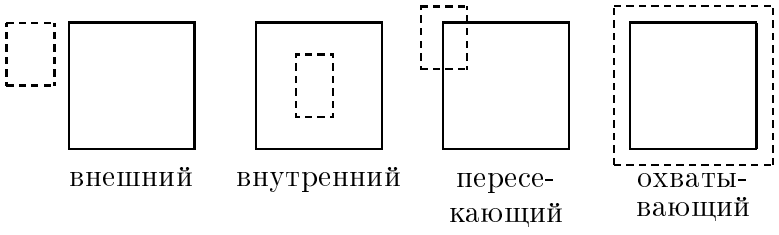


Рис. 0.10.1: Соотношения между окном экрана (сплошная рамка) и многоугольником (штриховая рамка)

В четырех случаях можно сразу принять решение о правилах закраски области экрана:

- все многоугольники сцены — внешние по отношению к окну. В этом случае окно закрашивается фоном;
- имеется всего один внутренний или пересекающий многоугольник. В этом случае все окно закрашивается фоном и затем часть окна, соответствующая внутреннему или пересекающему окну закрашивается цветом многоугольника;
- имеется единственный охватывающий многоугольник. В этом случае окно закрашивается его цветом.
- имеется несколько различных многоугольников и хотя бы один из них охватывающий. Если при этом охватывающий многоугольник расположен ближе остальных к наблюдателю, то окно закрашивается его цветом.

В любых других случаях процесс разбиения окна продолжается. Легко видеть, что при растре 1024×1024 и делении стороны окна пополам требуется не более 10 разбиений. Если достигнуто максимальное разбиение, но не обнаружено ни одного из приведенных выше четырех случаев, то для точки с центром в полученном минимальном окне (размером в пиксел) вычисляются глубины оставшихся многоугольников и закраску определяет многоугольник, наиболее близкий к наблюдателю. При этом для устранения лестничного эффекта можно выполнить дополнительные разбиения и закрасить пиксел с учетом всех многоугольников, видимых в минимальном окне.

Первые три случая идентифицируются легко. Последний же случай фактически сводится к поиску охватывающего многоугольника, перекрывающего все остальные многоугольники, связанные с окном. Проверка на такой многоугольник может быть выполнена следующим образом: в угловых точках окна вычисляются Z-координаты для всех многоугольников, связанных с окном. Если все четыре такие Z-координаты охватывающего многоугольника ближе к наблюдателю, чем все остальные, то окно закрашивается цветом соответствующего охватывающего многоугольника. Если же нет, то мы имеем сложный случай и разбиение следует продолжить.

Очевидно, что после разбиения окна охватывающие и внешние многоугольники наследуются от исходного окна. Поэтому необходимо проверять лишь внутренние и пересекающие многоугольники.

Из изложенного ясно, что важной частью алгоритма является определение расположения многоугольника относительно окна.

Проверка на то что многоугольник внешний или внутренний относительно окна для случая прямоугольных окон легко реализуется использованием прямоугольной оболочки многоугольника и сравнением координат. Для внутреннего многоугольника должны одновременно выполняться условия:

$$\mathrm{Xmin} \geq \mathrm{W} \quad \mathrm{Xmax} \leq \mathrm{W} \quad \mathrm{Ymin} \geq \mathrm{W} \quad \mathrm{Ymax} \leq \mathrm{W},$$

здесь Xmin,Xmax,Ymin,Ymax — ребра оболочки
Wл, Wп, Wн, Wв — ребра окна

Для внешнего многоугольника достаточно выполнение любого из следующих условий:

$$\mathrm{Xmin} > \mathrm{X}, \quad \mathrm{Xmax} < \mathrm{W}, \quad \mathrm{Ymin} > \mathrm{W}, \quad \mathrm{Ymax} < \mathrm{W}$$

Таким способом внешний многоугольник, охватывающий угол окна не будет идентифицирован как внешний (см. рис. 0.10.2).

Проверка на пересечение окна многоугольником может быть выполнена проверкой на расположение всех вершин окна по одну сторону от прямой, на которой расположено ребро мно-

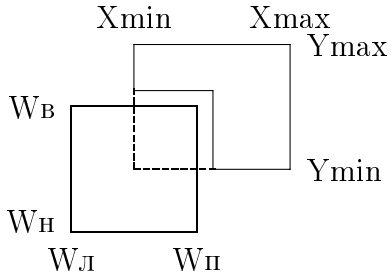
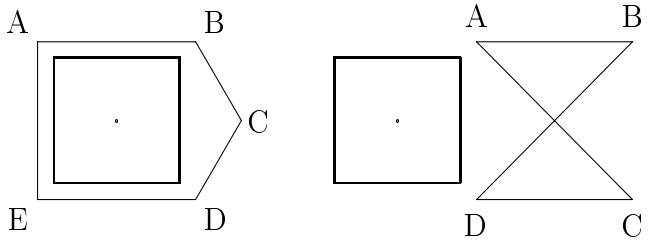


Рис. 0.10.2: Ошибочное определение внешнего многоугольника как пересекающего при использовании прямоугольной оболочки

гоугольника. Пусть ребро многоугольника задано точками P1(x1,y1,z1) и P2(x2,y2,z2), а очередная вершина окна задается точкой P3(x3,y3,z3). Векторное произведение вектора **P1P3** на вектор **P1P2**, равное (x3-x1)(y2-y1) — (y3-y1)(x2-x1) будет меньше 0, равно 0 или больше 0, если вершина лежит слева, на или справа от прямой P1P2. Если знаки различны, то окно и многоугольник пересекаются. Если же все знаки одинаковы, то окно лежит по одну сторону от ребра, т.е. многоугольник может быть либо внешним, либо охватывающим.

Вернемся к примеру 0.10.2. Такой многоугольник рассмотренными тестами не был идентифицирован ни как внутренний ни как пересекающий. Т.е. он может быть либо внешним, либо охватывающим. Для завершающей классификации может использоваться тест с подсчетом угла, рассматривавшийся ранее для определения нахождения точки внутри/вне многоугольника. В этом тесте вычисляется суммарный угол, на который повернется луч, исходящий из некоторой точки окна (обычно центра), при последовательном обходе вершин многоугольника.

Если суммарный угол равен 0, то многоугольник — внешний. Если же угол равен N×360°, то многоугольник охватывает окно N раз. Простейшая иллюстрация этого теста приведена на рис. 0.10.3.



а) многоугольник ABCDE — охватывающий Суммарный угол равен 360°

б) многоугольник ABCD — внешний Суммарный угол равен 0°

Рис. 0.10.3: Тест на охватывающий/внешний многоугольник

0.10.6 Построчный алгоритм Уоткинса

В алгоритмах построчного сканирования результирующее изображение генерируется построчно причем, подобно ранее рассмотренному алгоритму построчной заливки многоугольни-

ка, используется связность соседних растровых строк изображения. Отличие состоит в том, что учитываются все, а не один многоугольник.

Алгоритм работает в пространстве изображения с окном высотой в одну строку и шириной в экран, тем самым трехмерная задача сводится к двумерной.

Последовательность шагов алгоритма:

- построение списка ребер,
- построение списка многоугольников,
- построение списка активных ребер — создается таблица ребер, включающая все негоризонтальные ребра многоугольников, причем элементы таблицы по значению Y-координаты отсортированы по группам.

0.10.7 Алгоритм трассировки лучей

При рассмотрении этого алгоритма предполагается, что наблюдатель находится на положительной полуоси Z, а экран дисплея перпендикулярен оси Z и располагается между объектом и наблюдателем.

Удаление невидимых (скрытых) поверхностей в алгоритме трассировки лучей выполняется следующим образом:

- сцена преобразуется в пространство изображения,
- из точки наблюдения в каждый пиксел экрана проводится луч и определяется какие именно объекты сцены пересекаются с лучом,
- вычисляются и упорядочиваются по Z координаты точек пересечения объектов с лучом. В простейшем случае для непрозрачных поверхностей без отражений и преломлений видимой точкой будет точка с максимальным значением Z-координаты. Для более сложных случаев требуется сортировка точек пересечения вдоль луча.

Ясно, что наиболее важная часть алгоритма — процедура определения пересечения, которая в принципе выполняется Rx × Ry × N раз (здесь Rx,Ry — разрешение дисплея по X и Y, соответственно, а N — количество многоугольников в сцене).

Очевидно, что повышение эффективности может достигаться сокращением времени вычисления пересечений и избавлением от ненужных вычислений. Последнее обеспечивается использованием геометрически простой оболочки, объемлющей объект — если луч не пересекает оболочку, то не нужно вычислять пересечения с ним многоугольников, составляющих исследуемый объект.

При использовании прямоугольной оболочки определяется преобразование, совмещающее луч с осью Z. Оболочка подвергается этому преобразованию, а затем попарно сравниваются знаки Xmin с Xmax и Ymin с Ymax. Если они различны, то есть пересечение луча с оболочкой (см. рис. 0.10.4)

При использовании сферической оболочки для определения пересечения луча со сферой достаточно сосчитать расстояние от луча до центра сферы. Если оно больше радиуса, то пересечения нет. Параметрическое уравнение луча, проходящего через две точки P1(x1,y1,z1) и P2(x2,y2,z2), имеет вид:

$$P(t) = P1 + (P2 - P1) \times t$$

Минимальное расстояние от точки центра сферы P0(x0,y0,z0) до луча равно:

$$d^2 = (x - x0)^2 + (y - y0)^2 + (z - z0)^2$$

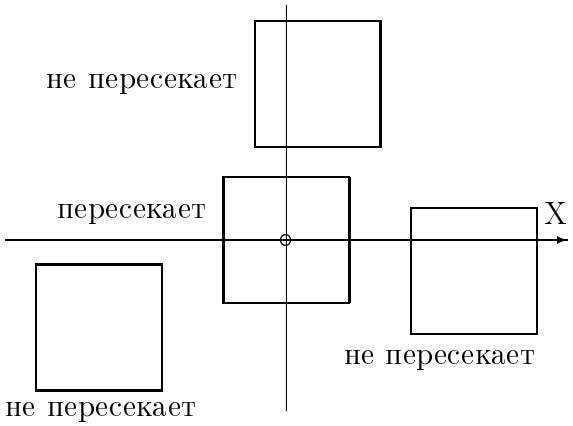


Рис. 0.10.4: Определение пересечения луча и оболочки

Этому соответствует значение t:

$$t = -\frac{(x2 - x1) \cdot (x1 - x0) + (y2 - y1) \cdot (y1 - y0) + (z2 - z1) \cdot (z1 - z0)}{(x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2}.$$

Если $d2 > R2$, то луч не пересекает объекты, заключенные в оболочку.

Дальнейшее сокращение расчетов пересечений основывается на использовании групп пространственно связанных объектов. Каждая такая группа окружается общей оболочкой. Получается иерархическая последовательность оболочек, вложенная в общую оболочку для всей сцены. Если луч не пересекает какую-либо оболочку, то из рассмотрения исключаются все оболочки, вложенные в нее и, следовательно, объекты. Если же луч пересекает некоторую оболочку, то рекурсивно анализируются все оболочки вложенные в нее.

Наряду с вложенными оболочками для сокращения расчетов пересечений используется отложенное вычисление пересечений с объектами. Если обнаруживается, что объект пересекается лучом, то он заносится в специальный список пересеченных. После завершения обработки всех объектов сцены объекты, попавшие в список пересеченных упорядочиваются по глубине. Заведомо невидимые отбрасываются а для оставшихся выполняется расчет пересечений и отображается точка пересечения наиболее близкая к наблюдателю.

Дополнительное сокращение объема вычислений может достигаться отбрасыванием нелицевых граней, учетов связности строк растрового разложения и т.д.

Для сокращения времени вычислений собственно пересечений предложено достаточно много алгоритмов, упрощающих вычисления для определенной формы задания поверхностей.

0.11 РЕАЛИСТИЧНОЕ ПРЕДСТАВЛЕНИЕ СЦЕН

Где нужен реализм:
• в конструировании,
• в архитектуре,
• в биологии и медицине,
• в науке (компьютерное моделирование),
• в масс-медиа,
• в тренажерах, играх.

Основные направления:
• синтез реалистичных изображений,
• реалистическое оживление синтезированных объектов.

С точки зрения приложений в науке и промышленности наиболее важно первое направление. Ключевая проблема — реалистическое представление освещенности:
• модели освещения, прозрачность, тени, фактура, • глобальная модель освещения с трассировкой лучей, • излучательность.

0.11.1 Модели освещения

Механизм диффузного и зеркального отражения света

Диффузное отражение света точечного источника от идеального рассеивателя определяется по закону Ламберта, согласно которому падающий свет рассеивается во все стороны с одинаковой интенсивностью. В этом случае освещенность точки пропорциональна доле ее площади, видимой от источника.

$$\mathrm{Ir} = \mathrm{Ip} \cdot \mathrm{Pd} \cdot \cos(\phi),$$

где Ir — интенсивность отраженного света, Ip — интенсивность точечного источника, $0 \leq \mathrm{Pd} \leq 1$ — коэффициент диффузного отражения, зависящий от материала поверхности и длины волны, $0 \leq \phi \leq \pi/2$ — угол между направлением света и нормалью к поверхности.

В реальных сценах, кроме света от точечных источников, присутствует и рассеянный свет, который упрощенно учитывается с помощью коэффициента рассеяния:

$$\mathrm{I} = \mathrm{Ir} \cdot \mathrm{Pr} + \mathrm{Ip} \cdot \mathrm{Pd} \cdot \cos(\phi),$$

где Ir — интенсивность рассеянного света, $0 \leq \mathrm{Pr} \leq 1$ — коэффициент диффузного отражения рассеянного света.

Субъективно достаточно реалистичный учет расстояния от центра проекции до объекта обеспечивается линейным затуханием:

$$\mathrm{I} = \mathrm{Ir} \cdot \mathrm{Pr} + \frac{\mathrm{Ip} \cdot \mathrm{Pd} \cdot \cos(\phi)}{\mathrm{d} + \mathrm{K}},$$

где d — расстояние от центра проекции до объекта, а K — произвольная константа.

При параллельной проекции, когда точка наблюдения находится в бесконечности, учет расстояния обеспечивается тем, что объект, ближайший к точке наблюдения, освещается полностью, далее расположенные — с уменьшенной освещенностью и в качестве d берется расстояние от объекта ближайшего к наблюдателю.

Свет, отраженный от идеального зеркала, виден только если угол между направлениями наблюдения и отражения равен нулю. Для неидеальных отражающих поверхностей используется модель Фонга [27]:

$$\mathrm{Is} = \mathrm{Ip} \cdot \mathrm{W}(\lambda, \phi) \cdot \cos^{\mathrm{n}}(\alpha),$$

где $\mathrm{W}(\lambda, \phi)$ — кривая отражения, зависящая от длины волны λ света источника и угла падения ϕ, $-\pi/1 \leq \alpha \leq \pi/2$ — угол между направлениями наблюдения и отражения, $1 \leq \mathrm{n} \leq 200$ — показатель степени, определяющий убывание интенсивности при изменении угла.

Часто $\mathrm{W}(\lambda, \phi)$ заменяется константой Ks, такой чтобы полученная картина была субъективно приемлема.

Суммарная модель освещения имеет вид:

$$\mathrm{I} = \mathrm{Ir} \cdot \mathrm{Pr} + \frac{\mathrm{Ip}}{\mathrm{d} + \mathrm{K}}(\mathrm{Pd} \cdot \cos(\phi) + \mathrm{W}(\lambda, \phi) \cdot \cos^{\mathrm{n}}(\alpha)).$$

Или при использовании вместо $\mathrm{W}(\lambda, \phi)$ константы Ks:

$$\mathrm{I} = \mathrm{Ir} \cdot \mathrm{Pr} + \frac{\mathrm{Ip}}{\mathrm{d} + \mathrm{K}}(\mathrm{Pd} \cdot \cos(\phi) + \mathrm{Ks} \cdot \cos^{\mathrm{n}}(\alpha)).$$

Если использовать нормированные вектора направлений падения $\mathbf{L}$, нормали $\mathbf{N}$, отражения $\mathbf{R}$ и наблюдения $\mathbf{V}$, то модель освещения для одного источника принимает вид:

$$\mathrm{I} = \mathrm{Ir} \cdot \mathrm{Pr} + \frac{\mathrm{Ip}}{\mathrm{d} + \mathrm{K}}(\mathrm{Pd} \cdot \mathbf{L} \cdot \mathbf{N} + \mathrm{Ks} \cdot (\mathbf{R} \cdot \mathbf{V})^{\mathrm{n}}).$$

Если источник света находится бесконечности, то для данного плоского многоугольника $\mathbf{L}\cdot\mathbf{N}$ равно константе, а $\mathbf{R}\cdot\mathbf{V}$ меняется в пределах многоугольника. Для поверхностей, представленных например в виде бикубических кусков, каждое произведение меняется в пределах куска. Так как эти вычисления требуется производить для каждого пиксела строки, то вычислительные затраты могут быть очень велики. Фонг предложил алгоритм пошагового вычисления по рассмотренной модели, существенно снижающий затраты.

Кроме эмпирической модели освещенности Фонга используются модели, представляющие отражающую поверхность в виде плоских микроскопических граней. Ориентации нормалей к граням относительно нормали к средней линии поверхности задаются некоторым распределением, например, Гаусса.

0.11.2 Модели закраски

Существует три основных способа закраски многоугольников: однотонная закраска, закраска с интерполяцией интенсивности и закраска с интерполяцией векторов нормали.

При однотонной закраске предполагается, что и источник света и наблюдатель находятся в бесконечности, поэтому произведения $\mathbf{L}\cdot\mathbf{N}$ и $\mathbf{R}\cdot\mathbf{V}$ постоянны. На изображении могут быть хорошо заметны резкие перепады интенсивности между различно закрашенными многоугольниками. Если многоугольники представляют собой результат аппроксимации криволинейной поверхности, то изображение недостаточно реалистично.

В методе закраски с интерполяцией интенсивности (метод Гуро) нормали в вершинах многоугольников вычисляются как результат усреднения нормалей ко всем полигональным граням,

которым принадлежит данная вершина. Используя значения нормалей, вычисляют интенсивности в вершинах по той или иной модели освещения. Эти значения затем используются для билинейной интерполяции: для данной строки сканирования вначале находят значения интенсивностей на ребрах, а затем линейно интерполируют между ними при закраске вдоль строки.

В методе закраски с интерполяцией нормали (метод Фонга) значение нормали вдоль строки интерполируется между значениями нормалей на ребрах для данной строки. Значения нормалей на ребрах получается как результат интерполирования между вершинами. Значения же нормалей в вершинах являются результатом усреднения, как и выше рассмотренном методе. Значение нормали для каждого из пикселов строки используется для вычислений по той или иной модели освещения.

0.11.3 Прозрачность

В простейшей модели прозрачности преломление не учитывается. При расчетах по такой модели могут использоваться любые алгоритмы удаления невидимых поверхностей, учитывающие порядок расположения многоугольников. При использовании построчных алгоритмов если передний многоугольник оказывается прозрачным, то определяется ближайший из оставшихся, внутри которых находится строка сканирования. Суммарная закраска определяется следующим образом:

$$I = k \cdot I + (1 - k) \cdot I,$$

где $0 \leq k \leq 1$ — характеризует прозрачность ближнего многоугольника. Если $k = 1$, то он непрозрачен. Если же $k = 0$, то ближний многоугольник полностью прозрачен; Iб — интенсивность для пиксела ближнего многоугольника, Iд — дальнего.

0.11.4 Тени

Простой способ определения объектов, попавших в тень и, следовательно, неосвещенных, аналогичен алгоритму удаления невидимых поверхностей: те объекты, которые невидимы из источника освещения, но видимы из точки зрения находятся в тени. На первом шаге в алгоритме с учетом тени определяются все многоугольники, видимые из точки освещения. Затем выполняется удаление поверхностей невидимых из точки зрения. При выполнении закраски многоугольника проверяется не закрыт ли он многоугольником, видимым из источника освещения. Если да, то в модели освещения учитываются (если надо) все три компоненты — диффузное и зеркальное отражения и рассеянный свет. Если же перекрытия нет, то закрашиваемый многоугольник находится в тени и надо учитывать только рассеянный свет.

0.11.5 Фактура

Решение в лоб — представление в виде соответствующего (очень большого) количества многоугольников мало приемлемо. Более практичное решение — “натягивание” массива узора, полученного в результате оцифровки изображения реальной поверхности на раскрашиваемую. При этом значения из массива узора используются для масштабирования диффузной компоненты в модели освещения.

Для устранения лестничного эффекта должны учитываться все элементы узора, затрагивающие обрабатываемый пиксел изображения.

Такой метод влияет на раскраску поверхности, но оставляет ее гладкой. Неровности могут моделироваться возмущениями нормали поверхности. Другой способ, используемый при синтезе картин — метод фрактальной геометрии.

0.11.6 Трассировка лучей

Метод трассировки лучей используется не только для удаления невидимых частей, но, в основном, для получения высокореалистичных изображений с учетом отражений и преломлений света.

Прямой трассировкой лучей называется процесс расчета освещения сцены с испусканием от всех источников лучей во всех направлениях. При попадании на какой-либо объект сцены луч света может преломившись уйти внутрь тела или отразившись далее продолжить прямолинейное распространение до попадания на следующий объект и так далее. Следовательно, каждая точка сцены может освещаться либо напрямую источником, либо отраженным светом. Часть лучей в конце концов попадет в глаз наблюдателя и сформирует в нем изображение сцены.

Понятно, что вычисления, необходимые для трассировки всех лучей для всех источников и поверхностей слишком объемисты. Причем существенный вклад в полученное изображение внесет лишь небольшая часть оттрассированных лучей.

Для избавления от излишних вычислений используется обратная трассировка, в которой вычисляются интенсивности только лучей, попавших в глаз наблюдателя. В простейшей реализации обратной трассировки отслеживаются лучи, проходящие из глаза наблюдателя через каждый пиксел экрана в сцену. На каждой поверхности сцены, на которую попадает луч, в общем случае формируются отраженный и преломленный лучи. Каждый из таких лучей отслеживается, чтобы определить пересекаемые поверхности. В результате для каждого пиксела строится дерево пересечений. Ветви такого дерева представляют распространение луча в сцене, а узлы — пересечения с поверхностями в сцене. Окончательная закраска определяется прохождением по дереву и вычислением вклада каждой пересеченной поверхности в соответствии с используемыми моделями отражения. При этом различают и обычно по-разному рассчитывают первичную освещенность, непосредственно получаемую от источников света, и вторичную освещенность, получаемую от других объектов.

0.11.7 Излучательность

Сцену можно представить как набор поверхностей, обменивающихся лучистой энергией. Большинство реальных поверхностей является диффузными отражателями, когда падающий луч отражается или рассеивается во всех направлениях полусферы, находящейся над отражающей поверхностью. Особый здесь случай — отражение Ламберта (идеальная диффузия). Метод излучательности описывает баланс энергетического равновесия в замкнутой системе. Предполагается что поверхности идеально диффузны, т.е. после отражения падающий луч пропадает. Излучательность отдельной поверхности включает самоизлучение и отраженный или пропущенный свет.

$E_i = \frac{}{\cdot\cdot\cdot}$ — эмиссия,

P_i — безразмерный коэффициент отражения,

B_i — полный уровень света, исходящего от поверхности

F_{ij} — форм-фактор — часть энергии, исходящая от одной поверхности и достигающая другой

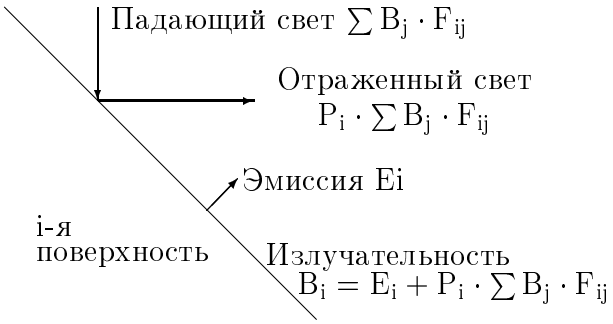


Рис. 0.11.1: Излучательность

Если поверхности сцены разделены на n элементов, имеющих постоянную излучательность, то взаимодействие потоков света в сцене описывается системой уравнений:

$$\begin{bmatrix} 1-P_1\cdot F_{11} & -P_1\cdot F_{12} & \dots & -P_1\cdot F_{1n} \\ 1-P_2\cdot F_{21} & -P_2\cdot F_{22} & \dots & -P_2\cdot F_{2n} \\ & \dots & & \\ 1-P_n\cdot F_{n1} & -P_n\cdot F_{n2} & \dots & -P_n\cdot F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \dots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \dots \\ E_n \end{bmatrix}.$$

Так как форм-фактор определяет часть энергии, исходящей от одной поверхности и достигающей другой, то сумма всех форм-факторов элемента меньше 1.

Эта система уравнений решается итерационным методом с выбором в качестве начального приближения для излучательности эмиссии поверхности:

$B_i^0 = E_i$,

и последующих приближений вида:

$B_i^{k+1} = E_i + P_i \sum_{j=1}^{n} F_{ij} B_j^k$.

После определения излучательности каждого фрагмента производится вычисление излучательностей вершин таким образом, чтобы при билинейной интерполяции в пределах каждого фрагмента была обеспечена непрерывность закраски на границах.

Для этого в каждом многоугольнике значения излучательности для внутренних относительно многоугольника вершин фрагментов вычисляются как среднее значение излучательностей фрагментов, окружающих вершину (рис. 0.11.2). Определения излучательности внешних вершин выполняют экстраполяцией средних значений в смежных внутренних вершинах (см. рис. 0.11.2).

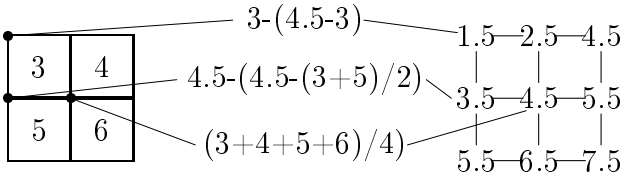
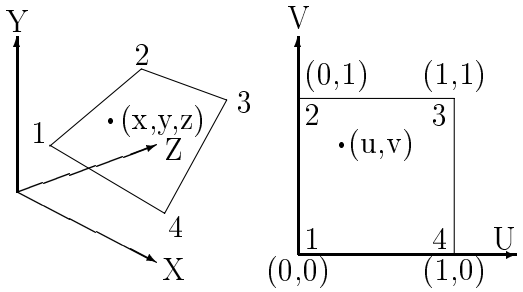


Рис. 0.11.2: Вычисление излучательности вершин

Формирование результирующего изображения выполняется после выбора требуемой точки наблюдения и проецирования сцены на картинную плоскость. Видно, что этот метод не зависит от точки наблюдения. При выводе определяется какой фрагмент отображается в каждом

пикселе экрана. При этом используются z-буфер и кадровый буфер. Пересечение линии, соединяющей глаз и пиксел с плоскостью фрагмента, позволяет найди координату точки, отображаемую в текущий пиксел. Координата пересечения (x, y, z) преобразуется в координаты (u, v) на фрагменте. После этого значения излучательностей вершин используются при билинейной интерполяции трех значений излучательностей в пределах каждого фрагмента (рис. 0.11.3).



$$\begin{aligned} B(u,v) = {} & (1-u)\cdot(1-v)\cdot B_1 + \\ & (1-u)\cdot B_2 + \\ & u\cdot v\cdot B_3 + \\ & u\cdot(1-v)\cdot B_4 \end{aligned}$$

Рис. 0.11.3: Билинейная интерполяция излучательностей по значениям в вершинах

Наиболее трудоемкая часть метода излучательности — вычисление форм факторов.

Для двух неперекрытых элементарных площадок i и j форм-фактор определяет часть световой энергии, исходящей из одной площадки на другую, и имеет вид:

$$F_{dAi,dAj} = \frac{\cos(\phi_i)\cdot\cos(\phi_j)}{\pi\cdot r^2}$$

где r — расстояние между элементами dA_i и dA_j, ϕ_i и ϕ_j — углы между нормалями к элементам и соединяющим их отрезком (рис. 0.11.4).

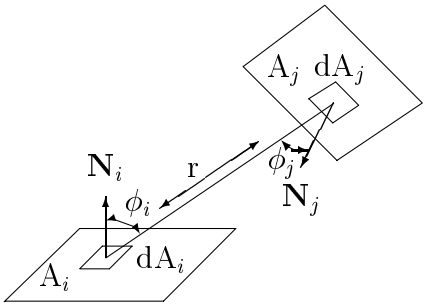


Рис. 0.11.4: Форм-фактор для двух элементарных площадок

Интегрируя по площади dA_j получаем форм-фактор конечного элемента площади A_j относительно элементарной площадки dA_i:

$$F_{dAi,Aj} = \int_{Aj} \frac{\cos(\phi_i)\cdot\cos(\phi_j)}{\pi\cdot r^2}\cdot dAj$$

Выражение для форм-фактора между двух конечных неперекрытых площадок определяется как средняя площадь:

$$F_{ij} = F_{Ai,Aj} = \frac{1}{A_i} \int_{Ai} \int_{Aj} \frac{\cos(\phi_i) \cdot \cos(\phi_j)}{\pi \cdot r^2} \cdot dA_j \cdot dA_i$$

Взаимное перекрытие может быть учтено с помощью функции H_{ij}, принимающей значение от 0 до 1 в зависимости от степени перекрытия площадки A_j площадкой A_i:

$$F_{ij} = F_{Ai,Aj} = \frac{1}{A_i} \int_{Ai} \int_{Aj} \frac{\cos(\phi_i) \cdot \cos(\phi_j)}{\pi \cdot r^2} \cdot H_{ij} \cdot dA_j \cdot dA_i$$

Вычисление этого интеграла, как правило, затруднено. Для его вычисления предложен т.н. алгоритм полукуба. Основная идея которого состоит в том, что если два любых фрагмента в пространстве после проецирования на полусферу заняли на ней одну и ту же площадь и место, то их формфакторы будут одинаковы. Это утверждение справедливо и при проецировании на любую другую окружающую поверхность, в том числе куб.

Куб строится таким образом, чтобы центр фрагмента, принимающего отраженные световые потоки, совпал с центром куба, а нормаль к фрагменту в этой точке примем за положительное направление оси Z. В этом случае полусфера заменяется верхней частью куба (полукубом). Разобьем плоскости полукуба на квадратные пикселы. Спроектируем все пространство на пять плоскостей полукуба при этом производится отсечение по пирамиде видимости с центром в центре куба. Если два фрагмента сцены проецируются на один и тот писел куба,

Литература

[1] Encarnacao J. Einfurung in die Graphische Datenverarbeiterung// Eurographics '89. Tutorial Notes 1. Hamburg, FRG, September 4-8, 1989. 122 s.

[2] Ньюмен У., Спрулл Р. Основы интерактивной машинной графики. Пер. с англ. М.: Мир, 1976.

[3] Роджерс Д. Алгоритмические основы машинной графики. Пер. с англ. М.: Мир, 1989. 512 с.

[4] Фоли Дж., вэн Дэм А. Основы интерактивной машинной графики: В 2-х книгах. Пер. с англ. М.: Мир, 1985.

[5] Антонофф М., Линдерхолм О. Лазерные принтеры// Компьютер Пресс, сборник N 1, 1989, с. 3-8.

[6] Введение в Автокад 11R: Метод.пособие/ НГТУ; Составители: Р.М.Сидорук, О.А.Соснина, И.М.Моисеенко. Н.Новгород, 1993. 181 с.

[7] Кречко Ю.А., Полищук В.В. Автокад. Курс практической работы. М.: "Диалог-МИФИ", 1994. 256 с.

[8] Вельтмандер П.В., Голубев В.М. Обучение автоматизации проектирования машиностроительного направления// Информатизация образования: Межвуз. сб. науч. тр./ под ред. В.Н.Врагова. Новосибирск: НГУ, 1994. С. 123–131.

[9] Винцюк Т.К. Системы речевого диалога// Материалы пятой школы-семинара "Интерактивные системы" (Кутаиси, 2-10 апреля 1983 г.). Тбилиси: Мецниереба, 1983, с. 16-22.

[10] Печатающие устройства персональных ЭВМ: Справочник// Под редакцией проф. И.М.Витенберга. М.: Радио и связь, 1992.

[11] Гилой В. Интерактивная машинная графика. Пер. с англ. М.: Мир, 1981.

[12] Грис Д. Конструирование компиляторов для цифровых вычислительных машин. Пер. с англ. М.: Мир, 1975. 544 с.

[13] Лисицин Б.Л. Низковольтные индикаторы: Справочник. М.: Радио и связь, 1985.

[14] Справочник по машинной графике в проектировании/ В.Е.Михайленко, В.А.Анпилогова, Л.А.Кириевский и др.: Под ред. В.Е.Михайленко. А.А.Лященко. Киев: Будівельник, 1984 . 184 с.

[15] Мячев А.А., Степанов В.Н. Персональные ЭВМ и микроЭВМ. Основы организации: Справочник/ Под ред. А.А.Мячева. М.: Радио и связь, 1991.

[16] Новаковский С.В. Цвет в цветном телевидении. М.: Радио и связь, 1988. 288 с.

[17] Павлидис Т. Алгоритмы машинной графики и обработки изображений. Пер. с англ. М.: Радио и связь, 1986.

[18] Прэтт У. Цифровая обработка изображений: Пер. с англ. в 2-х книгах. М.: Мир, 1982.

[19] Роджерс Д., Адамс Дж. Математические основы машинной графики. Пер. с англ. М.: Машиностроение, 1980.

[20] Сизых В.Г. Растровые дисплеи ряда Гамма. Новосибирск, 1985. 26 с. (Препринт ВЦ СО АН СССР; N 607).

[21] Ткаченко А.П. Цветное телевидение. Минск: Беларусь, 1981. 253 с.

[22] Фролов А.В., Фролов Г.В. Программирование видеоадаптеров CGA, EGA, VGA. М.: Диалог-МИФИ, 1992.

[23] Фостер Дж. Обработка списков. Пер. с англ. М.: Мир, 1974. 71 с.

[24] Холл П. Вычислительные структуры. Ведение в нечисленное программирование. Пер. с англ. М.: Мир, 1978. 214 с.

[25] Bresenham J.E. Algorithm for computer control of a digital plotter// IBM Systems Journal, vol. 4, No. 1, pp. 25–30, 1965.

[26] Bresenham J., A Linear Algoritm for Incremental Didgital Display of Circular Arcs, CACM, vol. 20, pp. 100-106, 1977.

[27] Bui-Tuong Phong. Illumination for Computer-Generated Pictures. Communication of the ASM, 18(6), June 1975, pp. 311–317.

[28] Clark, J.H. A VLSI geometry Processor for Graphics// IEEE Computer, 12(7).

[29] Cyrus M., Beck J. Generalized two- and threedimensional clipping// Computer and Graphics, Vol. 3, pp. 23–28, 1978.

[30] Hans Joseph, Max Mehl. Computer Graphics Hardware: Introduction and State of the Art// Eurographics '91. Tutorial Note 9. Viena, 2.–6. September 1991. Austria, Viena. 29 p.

[31] Fontenier Guy, Pascal Gros Pascal. Architectures of Graphic Processors for Interactive 2D Graphics// Computer Graphics Forum 7 (1988) 78-89.

[32] You-Dong Liang and Brian A. Barsky. A new concept and method for line clipping// ACM Transaction on Graphics, Vol. 3, No. 1, January 1984, pp. 1–22.

[33] Tina M. Nicholl, D.T.Lee and Robin A. Nicholl. An efficient new algoritm for 2-D line clipping: its development and analysis// Computer Graphics, V. 21, N. 4, July 1987, pp. 253–262.

[34] R.Pinkman, M.Novak, K.Guttag. Video-RAM exels at fast graphics// Electronics Design, pp. 161–171 (August 18 1983).

[35] H.-P. Seidel. PC Graphics Hardware // Eurographics '88. Tutorial/Cours 8. Nice, 12.–16. September 1988. France, Nice. 44 p.

[36] Smit A.R., Tint Fill, SIGGRAPH'79 Proceedings // Computer Graphics, Vol.13(2), Aug. 1979, pp. 276–283.

[37] Mark S. Sobkow, Paul Pospisil and Yee-Hong Yang. A Fast Two-Dimensional Line Clipping Algoritm via Line Encoding//Computer & Graphics, Vol. 11, No. 4, pp. 459–467, 1987.

[38] Robert F. Sproull and Ivan E. Sutherland. A Clipping Divider // AFIP Fall Joint Computer Conference. San Francisco, 1968.

[39] Stralunsfreier Flacbildschirm. MC, Die MikrocomputerZeitschrift. N 8, 1989, s. 66.

[40] Sutherland I.E., Hodgman G.W. Reentrant Polygon Clipping//Communications of the ACM, 17(1), pp. 32–42.

[41] Weiler K., Atherton P,. Hidden Surface Removal Using Polygon Area Sorting// SIGGGRAPH'77 Proceedings, Computer Graphics, Vol. 11, N. 2, pp. 214–222, 1977.

[42] Weiler K., Polygon Comparision Using a Graph Representation// SIGGGRAPH'80 Proceedings, Computer Graphics, Vol. 14, pp. 10–18, 1980.

0.12 Приложение 1. Процедуры преобразований

0.13 Приложение 2. Процедуры генерации отрезков

Здесь приведены тексты соответствующих процедур с пояснениями и тестовая программа. Процедуры позволяют генерировать вектора в любом квадранте с использованием алгоритмов несимметричного цифрового дифференциального анализатора и Брезенхема, а также построения ребра, ограничивающего заполненный многоугольник, модифицированным алгоритмом Брезенхема, уменьшающим лестничный эффект.

Предусмотрена возможность задания атрибутов формируемых отрезков — номер цвета и размер пиксела, используемого при формировании отрезков.

В тестовой программе предусмотрено, что при наличии SVGA-адаптера он может использоваться как в обычном режиме, так и в режиме до 1024x768 точек с 256 цветами.

```
/*------------------------------------------------ V_VECTOR.C
 * Подпрограммы генерации векторов
 */

#include <graphics.h>
#include <stdio.h>

#define PutMay putpixel

static int Pix_X= 3,    /* Размер пиксела по X  */
           Pix_Y= 3,    /* Размер пиксела по Y  */
           Pix_C= 64,   /* Нач. индекс цвета пиксела */
           Pix_V= 64;   /* Количество оттенков  */

/*------------------------------------------------- PutPixLn
 * Подпрограмма заносит "крупный" пиксел в позицию X,Y
 * Точка (0, 0) - левый верхний угол экрана.
 * Размеры пиксела задается фактическими параметрами x и y.
 * Номер оттенка пиксела задается фактическим параметром c.
 */

void PutPixLn (int x, int y, int c)
{  int  ii, jj, kk;
   x *= Pix_X;  y *= Pix_Y;
   ii= Pix_Y;
   while (--ii >= 0) {
      jj= Pix_X;  kk= x;
      while (--jj >= 0) PutMay (kk++, y, c);
      y++;
   }
}  /* PutPixLn */


/*------------------------------------------------- V_setlin
 * Устанавливает атрибуты построения линий:
```

```
 * размер элементарного пиксела, индекс цвета, кол-во оттенков
 * Если размер <= 0, то он не меняется
 * Если атрибут цвета < 0, то он не меняется
 */
void V_setlin (sizex, sizey, colorindex, colorvalue)
int  sizex, sizey, colorindex;
{
   if (sizex > 0) Pix_X= sizex;
   if (sizey > 0) Pix_Y= sizey;
   if (colorindex >= 0) Pix_C= colorindex;
   if (colorvalue >= 0) Pix_V= colorvalue;
}  /* V_setlin */
```

0.13.1 V_DDA — несимметричный ЦДА

```
/*-----------------------------------------------------  V_DDA
 * void V_DDA (int xn, int yn, int xk, int yk)
 *
 * Подпрограмма построения вектора из точки (xn,yn)
 * в точку (xk, yk) в первом квадранте методом
 * несимметричного цифрового дифференциального анализатора
 * с использованием только целочисленной арифметики.
 *
 * Обобщение на другие квадранты труда не составляет.
 *
 * Построение ведется от точки с меньшими  координатами
 * к точке с большими координатами с единичным шагом по
 * координате с большим приращением.
 *
 * Отдельно выделяется случай вектора с dx == dy
 *
 * Всего надо выдать пикселы в dx= xk - xn + 1 позиции
 * по оси X и в dy= yk - yn + 1 позиции по оси Y.
 *
 * Для определенности рассмотрим случай dx > dy
 *
 * При приращении X-координаты на 1 Y-координата должна
 * увеличиться на величину меньшую единицы и равную dy/dx.
 *
 * После того как Y-приращение станет больше или равно 1.0,
 * то Y-координату пиксела надо увеличить на 1, а из
 * накопленного приращения вычесть 1.0 и продолжить построения
 * Т.е. приращение Y на 1 выполняется при условии:
 * dy/dx + dy/dx + ... + dy/dx >= 1.0
 * Т.к. вычисления в целочисленной арифметике быстрее, то
 * умножим на dx обе части и получим эквивалентное условие:
```

```
 * dy + dy + ... + dy >= dx
 *
 * Эта схема и реализована в подпрограмме.
 *
 * При реализации на ассемблере можно избавиться от
 * большинства операторов внутри цикла while.
 * Для этого перед циклом надо домножить dy на величину,
 * равную 65536/dx.
 * Тогда надо увеличивать Y на 1 при признаке переноса
 * после вычисления s, т.е. операторы
 *       s= s + dy;
 *       if (s >= dx) { s= s - dx;  yn= yn + 1; }
 * заменяются командами ADD и ADC
 *
 */

void V_DDA (xn, yn, xk, yk)
int  xn, yn, xk, yk;
{  int  dx, dy, s;

/* Упорядочивание координат и вычисление приращений */
   if (xn > xk) {
      s= xn;  xn= xk;  xk= s;
      s= yn;  yn= yk;  yk= s;
   }
   dx= xk - xn;  dy= yk - yn;

/* Занесение начальной точки вектора */
   PutPixLn (xn, yn, Pix_C);

   if (dx==0 && dy==0) return;

/* Вычисление количества позиций по X и Y */
   dx= dx + 1;  dy= dy + 1;

/* Собственно генерация вектора */
   if (dy == dx) {                 /* Наклон == 45 градусов */
      while (xn < xk) {
         xn= xn + 1;
         PutPixLn (xn, xn, Pix_C);
      }
   } else if (dx > dy) {           /* Наклон <  45 градусов */
      s= 0;
      while (xn < xk) {
         xn= xn + 1;
```

```
         s= s + dy;
         if (s >= dx) { s= s - dx;  yn= yn + 1; }
         PutPixLn (xn, yn, Pix_C);
      }
   } else {                        /* Наклон >  45 градусов */
      s= 0;
      while (yn < yk) {
         yn= yn + 1;
         s= s + dx;
         if (s >= dy) { s= s - dy;  xn= xn + 1; }
         PutPixLn (xn, yn, Pix_C);
      }
   }
}  /* V_DDA */
```

0.13.2 V_Bre — алгоритм Брезенхема

```
/*-------------------------------------------------------  V_Bre
 * void V_Bre (int xn, int yn, int xk, int yk)
 *
 * Подпрограмма иллюстрирующая построение вектора из точки
 * (xn,yn) в точку (xk, yk) методом Брезенхема.
 *
 * Построение ведется от точки с меньшими  координатами
 * к точке с большими координатами с единичным шагом по
 * координате с большим приращением.
 *
 * В общем случае исходный вектор проходит не через вершины
 * растровой сетки, а пересекает ее стороны.
 * Пусть приращение по X больше приращения по Y и оба они > 0.
 * Для очередного значения X нужно выбрать одну двух ближайших
 * координат сетки по Y.
 * Для этого проверяется как проходит  исходный  вектор - выше
 * или ниже середины расстояния между ближайшими значениями Y.
 * Если выше середины,  то Y-координату  надо  увеличить на 1,
 * иначе оставить прежней.
 * Для этой проверки анализируется знак переменной s,
 * соответствующей разности между истинным положением и
 * серединой расстояния между ближайшими Y-узлами сетки.
 */

void V_Bre (xn, yn, xk, yk)
int  xn, yn, xk, yk;
{  int  dx, dy, s, sx, sy, kl, swap, incr1, incr2;

/* Вычисление приращений и шагов */
```

```
   sx= 0;
   if ((dx= xk-xn) < 0) {dx= -dx; --sx;} else if (dx>0) ++sx;
   sy= 0;
   if ((dy= yk-yn) < 0) {dy= -dy; --sy;} else if (dy>0) ++sy;
/* Учет наклона */
   swap= 0;
   if ((kl= dx) < (s= dy)) {
      dx= s;  dy= kl;  kl= s; ++swap;
   }
   s= (incr1= 2*dy)-dx; /* incr1 - констан. перевычисления */
                        /* разности если текущее s < 0  и  */
                        /* s - начальное значение разности */
   incr2= 2*dx;         /* Константа для перевычисления    */
                        /* разности если текущее s >= 0    */
   PutPixLn (xn,yn,Pix_C); /* Первый  пиксел вектора       */
   while (--kl >= 0) {
      if (s >= 0) {
         if (swap) xn+= sx; else yn+= sy;
         s-= incr2;
      }
      if (swap) yn+= sy; else xn+= sx;
      s+=  incr1;
      PutPixLn (xn,yn,Pix_C); /* Текущая  точка  вектора   */
   }
}  /* V_Bre */
```

0.13.3 V_BreM — модифицированный алгоритм Брезенхема

```
/*------------------------------------------------- V_BreM
 * void V_BreM (int xn, int yn, int xk, int yk)
 *
 * Подпрограмма иллюстрирующая построение ребра залитого
 * многоугольника из точки (xn,yn) в точку (xk,yk)
 * модифицированным методом Брезенхема.
 *
 * Строки многоугольника от занесенного пиксела границы до xk
 * заполняются оттенком с максимальным номером.
 */

void V_BreM (xn, yn, xk, yk)
int  xn, yn, xk, yk;
{  int  dx, dy, sx, sy, kl, swap;
   long incr1, incr2;
   long s;              /* Текущее значение ошибки  */
   long s_max;          /* Макс значение ошибки     */
   int  color_tek;      /* Текущий номер оттенка    */
```

```
   int  xt;

/* Вычисление приращений и шагов */
   sx= 0;
   if ((dx= xk-xn) < 0) {dx= -dx; --sx;} else if (dx>0) ++sx;
   sy= 0;
   if ((dy= yk-yn) < 0) {dy= -dy; --sy;} else if (dy>0) ++sy;
/* Учет наклона */
   swap= 0;
   if ((kl= dx) < (s= dy)) {dx= s;  dy= kl;  kl= s; ++swap;}
   s= (long)dx*(long)Pix_V; /* Начальное значение ошибки    */
   incr1= 2l*(long)dy       /* Конст. перевычисления ошибки */
            *(long)Pix_V;   /* если текущее s < s_max       */
   incr2= 2l*s;             /* Конст. перевычисления ошибки */
                            /* если текущее s >= s_max      */
   s_max= incr2 - incr1;    /* Максимальное значение ошибки */
   color_tek= Pix_V;        /* Яркость стартового пиксела   */
   if (dx)color_tek=(int)((((long)Pix_V*(long)dy)/(long)dx)/2l);
   PutPixLn (xn, yn, Pix_C+color_tek);      /* 1-й пиксел */
   while (--kl >= 0) {
      if (s >= s_max) {
         if (swap) xn+= sx; else yn+= sy;
         s-= incr2;
      }
      if (swap) yn+= sy; else xn+= sx;
      s+=  incr1;
      color_tek= Pix_V;
      if (dx) color_tek= s / dx /2;
      PutPixLn (xn,yn,Pix_C+color_tek);     /* Тек.пиксел */
/* Однотонная закраска строки многоугольника макс цветом  */
      xt= xn;
      while (++xt <= xk) PutPixLn (xt,yn,Pix_C+Pix_V-1);
   }
}  /* V_BreM */
```

0.13.4 T_VECTOR — тестовая программа генерации векторов

```
/*================================================ T_VECTOR.C
 * ТЕСТ ГЕНЕРАЦИИ ВЕКТОРОВ
 *
 * Строит вектора из точки Xn,Yn в заданную
 * Программа запрашивает ввод четырех чисел:
 * mode  = -2 - прекращение работы
 *         -1 - очистка экрана
 *          0 - вывод сетки
 *          1-7 построение вектора:
```

```
 *            1pp == 1 - по алгоритму ЦДА
 *            2pp == 1 - по алгоритму Брезенхема
 *            3pp == 1 - по модифиц. алгоритму Брезенхема
 *         иное значение - замена Xn,Yn на введенные Xk,Yk
 * atrib - атрибуты построения в виде десятичного числа
 *         из 8 цифр - PPCCCVVV:
 *         PP  - размер элементарного пиксела
 *         CCC - начальный номер оттенка
 *         VVV - количество оттенков
 * Xk    - конечная координата вектора
 * Yk
 */

#include "V_VECTOR.C"

#define MODE_256 1 /* 0/1 - обычный VGA/SVGA режим */

#if MODE_256
#   include "V_SVGA.C"
#endif

#include <conio.h>
#include <graphics.h>
#include <stdio.h>
#include <stdlib.h>

/*-------------------------------------------------------- Grid
 * Строит сетку 10*10
 */

void Grid (int col)
{  int Xn,Yn,Xk,Yk;
   setcolor (col);
   Xn= 0;  Xk= getmaxx();
   Yn= 0;  Yk= getmaxy();
   while (Xn <= Xk) {line (Xn,Yn,Xn,Yk); Xn+= 10; }
   Xn= 0;
   while (Yn <= Yk) {line (Xn,Yn,Xk,Yn); Yn+= 10; }
}  /* Grid */

/*---------------------------------------- MAIN T_VECTOR.C */

void main (void)
{
   int  ii, jj,
```

```
        mode=1,           /* Режим работы              */
        Xn=0,Yn=0,        /* Координаты начала отрезка */
        Xk,Yk,            /* Координаты конца отрезка  */
        fon,              /* Индекс цвета фона         */
        col_beg, col_val, /* Атрибуты пикселов         */
        xpix, ypix,
        colgrid,          /* Цвет сетки                */
        col_lin= 200,     /* Цвет "точного" отрезка    */
        col_Bre= 201,     /* Цвет построения для ЦДА   */
        col_DDA= 202;     /* Цвет построения для Брезенхема */
   int  gdriver= DETECT, gmode;
   long atrib=5064064l,la;/* Размер пиксела*100+цвета  */

#if MODE_256
   V_ini256 (&gdriver, &gmode, "");
   jj= getmaxcolor();
   for (ii=0; ii<=jj; ++ii)     /* Ч/б палитра    */
      setrgbpalette (ii, ii, ii, ii);
   atrib=5064064l;              /* Пиксел 5x5, нач цвет=64*/
   colgrid= 170;                /* Цвет сетки */
   fon= 140;
   setrgbpalette(7,255,255,255);/* Цвет для printf */
#else
   initgraph (&gdriver, &gmode, "");
   atrib= 5000016l;             /* Пиксел 5x5, нач цвет=0*/
   colgrid= 9;
   fon= 8;
#endif

   setbkcolor(fon);                     /* Очистка экрана */
   cleardevice();
   Xk= getmaxx(); Yk= getmaxy();

   Grid (colgrid);

/* Цвет для построения алгоритмом ЦДА */
   setrgbpalette(col_lin,63, 0,0);  /* Цвет точного отрезка  */
   setrgbpalette(col_DDA,63,63,0);  /* Цвет для ЦДА          */
   setrgbpalette(col_Bre,00,63,0);  /* Цвет для Брезенхема   */

   for (;;) {
      gotoxy (1, 1);
      printf("                                                ");
      printf("              \r");
      printf("mode atrib Xk Yk= (%d %ld %d %d) ? ",
```

```
               mode, atrib, Xk, Yk);
      scanf ("%d%ld%d%d", &mode, &atrib, &Xk, &Yk);
      xpix= ypix= atrib / 10000001;
      la= atrib % 10000001;
      col_beg= la / 1001;
      col_val= la % 1001;
      if (mode == -2) goto konec; else
      if (mode == -1) cleardevice(); else
      if (!mode) Grid (colgrid); else
      if (mode & 7) {
        if (mode & 1) {
           V_setlin (xpix, ypix, col_DDA, 1);
           V_DDA (Xn, Yn, Xk, Yk);
/* Построение "точного" отрезка */
           setcolor (col_lin);
           line (Xn, Yn, Xk*xpix, Yk*ypix);
        }
        if (mode & 2) {
           V_setlin (xpix, ypix, col_Bre, 1);
           V_Bre (Xn, Yn+3, Xk, Yk+3);
/* Построение "точного" отрезка */
           setcolor (col_lin);
           line (Xn, (Yn+3)*ypix, Xk*xpix, (Yk+3)*ypix);
        }
        if (mode & 4) {
           V_setlin (xpix, ypix, col_beg, col_val);
           V_BreM (Xn, Yn+6, Xk, Yk+6);
/* Построение "точного" отрезка */
           setcolor (col_lin);
           line (Xn, (Yn+6)*ypix, Xk*xpix, (Yk+6)*ypix);
        }
      } else {
         Xn= Xk;  Yn= Yk;
      }
   }
konec:
   closegraph();
}  /* main */
```

0.14 Приложение 3. Процедуры фильтрации

В данном приложении содержатся процедуры поддержки низкочастотной фильтрации растровых изображений и процедуры усреднения растровых изображений с понижением разрешения, а также тестовая программа демонстрирующая их работу.

Всего представлены две процедуры низкочастотной фильтрации V_fltr0 и V_fltr1, предназначенные для обработки изображений прямо в видеопамяти и с построчной буферизацией в оперативной памяти, соответственно. Последняя при модификации вспомогательных процедур доступа к изображению может обрабатывать картины, находящиеся в файле на внешнем носителе или просто в оперативной памяти.

Аналогично, представлены две процедуры усреднения изображения с понижением разрешения — V_fltr2 и V_fltr3.

При фильтрации и усреднении может использоваться одна из пяти предусмотренных масок фильтрации.

```
/*================================================ V_FILTR.C
 * В файле V_FILTR.C содержатся процедуры
 * поддержки фильтрации изображений:
 *
 * GetStr, PutStr - служебные
 *
 * V_fltr0 - фильтрует изображение в прямоугольной области,
 *           работая прямо с видеопамятью
 * V_fltr1 - фильтрует изображение в прямоугольной области,
 *           работая с буферами строк
 * V_fltr2 - усредняет картину по маске с понижением
 *           разрешения, работая прямо с видеопамятью
 * V_fltr3 - усредняет картину по маске с понижением
 *           разрешения, работая с буферами строк
 */

#include <alloc.h>

#define GetMay getpixel
#define PutMay putpixel

static int
   Mask0[]= {1,1, 1,1 },
   Mask1[]= {1,1,1, 1,1,1, 1,1,1 },
   Mask2[]= {1,1,1, 1,2,1, 1,1,1 },
   Mask3[]= {1,2,1, 2,4,2, 1,2,1 },
   Mask4[]= {1,1,1,1,
             1,1,1,1,
             1,1,1,1,
             1,1,1,1 },
   Mask5[]= {1,2, 3, 4, 3,2,1,
             2,4, 6, 8, 6,4,2,
```

```
            3,6, 9,12, 9,6,5,
            4,8,12,16,12,8,4,
            3,6, 9,12, 9,6,5,
            2,4, 6, 8, 6,4,2,
            1,2, 3, 4, 3,2,1 },
   Mask_ln[]= {2, 3, 3, 3, 4, 7},   /* Размер маски    */
   Mask_st[]= {2, 2, 2, 2, 4, 4},   /* Шаг усреднения  */
   Mask_vl[]= {4, 9,10,16,16,256},  /* Сумма элементов */
   *Mask_bg[]={                    /* Адреса начал    */
      Mask0,Mask1,Mask2,Mask3,Mask4,Mask5
   };

/*------------------------------------------------------ GetStr
 * Запрашивает фрагмент растровой строки из видеопамяти
 */
static void GetStr (st, Yst, Xn, Xk)
char *st;  int Yst, Xn, Xk;
{ while (Xn <= Xk) *st++= GetMay (Xn++, Yst); }


/*------------------------------------------------------ PutStr
 * Записывает фрагмент растровой строки в видеопамять
 */
static void PutStr (st, Yst, Xn, Xk)
char *st;  int Yst, Xn, Xk;
{while (Xn <= Xk) PutMay (Xn++, Yst, *st++); }


/*------------------------------------------------------ V_fltr0
 * Фильтрует изображение в прямоугольной области,
 * работая прямо с видеопамятью
 * msknum              = 0-5 - номер маски фильтра
 * Xn_source,Yn_source - окно исходного изображения
 * Xk_source,Xk_source
 * Xn_target,Yn_target - верхний левый угол результата
 */
void V_fltr0 (msknum,Xn_source,Yn_source,Xk_source,Yk_source,
              Xn_target,Yn_target)
int  msknum,Xn_source,Yn_source,Xk_source,Yk_source,
     Xn_target,Yn_target;
{
   char *plut;          /* Указатель палитры */
   int  *pi;            /* Тек указат маски  */
   int  pixel;          /* Пиксел исх изображения */
   int  *Maska,         /* Указатель маски */
        Mask_Y,Mask_X,  /* Размеры маски   */
```

```
        X_centr,Y_centr,/* Центр маски     */
        Mask_sum,       /* Сумма элементов */
        Xk,             /* Предельные положения маски */
        Yk,             /* в исходной области */
        s, sr, sg, sb,  /* Скаляры для суммир в маской   */
        ii, jj,
        Xt, Yt;

/* Запрос параметров маски */
   Maska=  Mask_bg[msknum];             /* Указатель маски */
   Mask_Y= Mask_X= Mask_ln[msknum];     /* Размеры маски   */
   X_centr= Mask_X / 2;                 /* Центр маски     */
   Y_centr= Mask_Y / 2;
   Mask_sum= Mask_vl[msknum];           /* Сумма элементов */

/* Предельные положения маски в исходной области */
   Xk= Xk_source+1-Mask_X;
   Yk= Yk_source+1-Mask_Y;

/*------- Фильтрация с прямой работой с видеопамятью -------*/

   for (Yt= Yn_source; Yt<=Yk; ++Yt) {
      for (Xt=Xn_source; Xt<=Xk; ++Xt) {
         pi= Maska; sr=0; sg=0; sb=0;   /* Суммированные RGB*/
         for (ii=0; ii<Mask_Y; ++ii)
            for (jj=0; jj<Mask_X; ++jj) {
               pixel= GetMay (Xt+jj, Yt+ii);
               plut= &V_pal256[pixel][0];
               s= *pi++;                /* Элемент маски */
               sr+= (s * *plut++);      /* Суммирование  */
               sg+= (s * *plut++);      /* по цветам с   */
               sb+= (s * *plut++);      /* весами маски  */
            }
         sr /= Mask_sum; sg /= Mask_sum;  sb /= Mask_sum;
/* Поиск элемента ТЦ, наиболее подходящего для данных R,G,B */
         ii= V_clrint (sr, sg, sb);
         PutMay (Xn_target+(Xt-Xn_source)+X_centr,
                 Yn_target+(Yt-Yn_source)+Y_centr, ii);
      }
   }
}  /* V_fltr0 */


/*---------------------------------------------------- V_fltr1
 * Фильтрует изображение в прямоугольной области,
 * работая с буферами строк
```

```
 * msknum              = 0-5 - номер маски фильтра
 * Xn_source,Yn_source - окно исходного изображения
 * Xk_source,Xk_source
 * Xn_target,Yn_target - верхний левый угол результата
 */
void V_fltr1 (msknum,Xn_source,Yn_source,Xk_source,Yk_source,
              Xn_target,Yn_target)
int  msknum,Xn_source,Yn_source,Xk_source,Yk_source,
     Xn_target,Yn_target;
{
   char *plut;          /* Указатель палитры */
   int  *pi;            /* Тек указат маски  */
   int  pixel;          /* Пиксел исх изображения */
   int  *Maska,         /* Указатель маски */
        Mask_Y,Mask_X,  /* Размеры маски   */
        X_centr,Y_centr,/* Центр маски     */
        Mask_sum,       /* Сумма элементов */
        Xk,             /* Предельные положения маски */
        Yk,             /* в исходной области */
        Dx_source,      /* Размер строки исх изображения */
        Ystr,           /* Y тек читаемой строки изображ */
        s, sr, sg, sb,  /* Скаляры для суммир в маской   */
        ii, jj,
        Xt, Yt;
   char *ps, *sbuf, *pt, *tbuf, *ptstr[8];

   Dx_source= Xk_source-Xn_source+1;

/* Запрос параметров маски */
   Maska=  Mask_bg[msknum];             /* Указатель маски */
   Mask_Y= Mask_X= Mask_ln[msknum];     /* Размеры маски   */
   X_centr= Mask_X / 2;                 /* Центр маски     */
   Y_centr= Mask_Y / 2;
   Mask_sum= Mask_vl[msknum];           /* Сумма элементов */

/* Предельные положения маски в исходной области */
   Xk= Xk_source+1-Mask_X;
   Yk= Yk_source+1-Mask_Y;

/* Заказ буферов */
   if ((sbuf= malloc (Dx_source * Mask_Y)) == NULL) goto all;
   if ((tbuf= malloc (Dx_source)) == NULL)
      goto fr_sbuf;

/*------- Фильтрация с использованием буферов строк --------*/
```

```
/* Подготовка массива указателей на строки
 *  ptstr[0] --> последняя строка
 *  ptstr[1] --> строка 0
 *  ptstr[2] --> строка 1
 *  и т.д.
 */
   ps= sbuf;  ii= Mask_Y;  jj= 1;
   do {
      ptstr[jj]= ps;  ps+= Dx_source;
      if (++jj == Mask_Y) jj= 0;
   } while (--ii > 0);

/* Начальное чтение Mask_Y - 1 строк */
   Ystr= Yn_source;
   for (ii=1; ii<Mask_Y; ++ii)
      GetStr (ptstr[ii], Ystr++, Xn_source, Xk_source);

   for (Yt= Yn_source; Yt<=Yk; ++Yt) {

/* Запрос следующей строки и циклический сдвиг указателей */
      GetStr (ps= ptstr[0], Ystr++, Xn_source, Xk_source);
      jj= Mask_Y-1;
      for (ii=0; ii<jj; ++ii) ptstr[ii]= ptstr[ii+1];
      ptstr[jj]= ps;

      pt= tbuf;
      for (Xt=Xn_source; Xt<=Xk; ++Xt) {
         pi= Maska; sr=0; sg=0; sb=0;   /* Суммированные RGB*/
         for (ii=0; ii<Mask_Y; ++ii) {
            ps= ptstr[ii] + (Xt-Xn_source);
            for (jj=0; jj<Mask_X; ++jj) {
               plut= &V_pal256[*ps++ & 255][0];
               s= *pi++;                /* Элемент маски */
               sr+= (s * *plut++);      /* Суммирование  */
               sg+= (s * *plut++);      /* по цветам с   */
               sb+= (s * *plut++);      /* весами маски  */
            }
         }
         sr /= Mask_sum; sg /= Mask_sum;  sb /= Mask_sum;
/* Поиск элемента ТЦ, наиболее подходящего для данных R,G,B */
         *pt++= V_clrint (sr, sg, sb);
      }
      PutStr (tbuf,                     /* Запись строки */
              Yn_target + Y_centr + (Yt-Yn_source) ,
```

```
              Xn_target + X_centr,
              Xn_target + X_centr + (--pt - tbuf));
   }
   free (tbuf);
fr_sbuf:
   free (sbuf);
all:;
}  /* V_fltr1 */


/*------------------------------------------------- V_fltr2
 * Усредняет картину по маске с понижением разрешения,
 * работая прямо с видеопамятью
 * msknum              = 0-5 - номер маски фильтра
 * Xn_source,Yn_source - окно исходного изображения
 * Xk_source,Xk_source
 * Xn_target,Yn_target - верхний левый угол результата
 */
void V_fltr2 (msknum,Xn_source,Yn_source,Xk_source,Yk_source,
              Xn_target,Yn_target)
int  msknum,Xn_source,Yn_source,Xk_source,Yk_source,
     Xn_target,Yn_target;
{
   char *plut;          /* Указатель палитры */
   int  *pi;            /* Тек указат маски  */
   int  pixel;          /* Пиксел исх изображения */
   int  *Maska,         /* Указатель маски */
        Mask_Y,Mask_X,  /* Размеры маски   */
        X_centr,Y_centr,/* Центр маски     */
        Mask_sum,       /* Сумма элементов */
        Xk,             /* Предельные положения маски */
        Yk,             /* в исходной области */
        s, sr, sg, sb,  /* Скаляры для суммир в маской   */
        Xr,Yr,          /* Координаты пиксела результата */
        Sm,             /* Сдвиг маски для обраб след точки */
        ii, jj,
        Xt, Yt;

/* Запрос параметров маски */
   Maska=  Mask_bg[msknum];             /* Указатель маски */
   Mask_Y= Mask_X= Mask_ln[msknum];     /* Размеры маски   */
   X_centr= Mask_X / 2;                 /* Центр маски     */
   Y_centr= Mask_Y / 2;
   Mask_sum= Mask_vl[msknum];           /* Сумма элементов */

/* Предельные положения маски в исходной области */
```

```
   Xk= Xk_source+1-Mask_X;
   Yk= Yk_source+1-Mask_Y;

   Yt= Yn_source;
   Yr= Yn_target+Y_centr;
   Sm= Mask_st[msknum];                 /* Шаг усреднения*/
   while (Yt <= Yk) {
      Xt=Xn_source;  Xr= Xn_target+X_centr;
      while (Xt <= Xk) {
         pi= Maska; sr=0; sg=0; sb=0;   /* Суммированные RGB*/
         for (ii=0; ii<Mask_Y; ++ii)
            for (jj=0; jj<Mask_X; ++jj) {
               pixel= GetMay (Xt+jj, Yt+ii);
               plut= &V_pal256[pixel][0];
               s= *pi++;                /* Элемент маски */
               sr+= (s * *plut++);      /* Суммирование  */
               sg+= (s * *plut++);      /* по цветам с   */
               sb+= (s * *plut++);      /* весами маски  */
            }
         sr /= Mask_sum; sg /= Mask_sum;  sb /= Mask_sum;
/* Поиск элемента ТЦ, наиболее подходящего для данных R,G,B */
         ii= V_clrint (sr, sg, sb);
         PutMay (Xr++, Yr, ii);
         Xt+= Sm;
      }
      Yt+= Sm;  ++Yr;
   }
}  /* V_fltr2 */


/*------------------------------------------------- V_fltr3
 * Усредняет картину по маске с понижением разрешения,
 * работая с буферами строк
 * msknum              = 0-5 - номер маски фильтра
 * Xn_source,Yn_source - окно исходного изображения
 * Xk_source,Xk_source
 * Xn_target,Yn_target - верхний левый угол результата
 */
void V_fltr3 (msknum,Xn_source,Yn_source,Xk_source,Yk_source,
              Xn_target,Yn_target)
int  msknum,Xn_source,Yn_source,Xk_source,Yk_source,
     Xn_target,Yn_target;
{
   char *plut;          /* Указатель палитры */
   int  *pi;            /* Тек указат маски  */
   int  pixel;          /* Пиксел исх изображения */
```

```
   int  *Maska,         /* Указатель маски */
        Mask_Y,Mask_X,  /* Размеры маски   */
        X_centr,Y_centr,/* Центр маски     */
        Mask_sum,       /* Сумма элементов */
        Xk,             /* Предельные положения маски */
        Yk,             /* в исходной области */
        Dx_source,      /* Размер строки исх изображения */
        s, sr, sg, sb,  /* Скаляры для суммир в маской   */
        Xr,Yr,          /* Координаты пиксела результата */
        Sm,             /* Сдвиг маски для обраб след точки */
        ii, jj,
        Xt, Yt;
   char *ps, *sbuf, *pt, *tbuf, *ptstr[8];


   Dx_source= Xk_source-Xn_source+1;

/* Запрос параметров маски */
   Maska=  Mask_bg[msknum];             /* Указатель маски */
   Mask_Y= Mask_X= Mask_ln[msknum];     /* Размеры маски   */
   X_centr= Mask_X / 2;                 /* Центр маски     */
   Y_centr= Mask_Y / 2;
   Mask_sum= Mask_vl[msknum];           /* Сумма элементов */

/* Предельные положения маски в исходной области */
   Xk= Xk_source+1-Mask_X;
   Yk= Yk_source+1-Mask_Y;

/* Заказ буферов */
   if ((sbuf= malloc (Dx_source * Mask_Y)) == NULL) goto all;
   if ((tbuf= malloc (Dx_source/Mask_st[msknum]+16)) == NULL)
      goto fr_sbuf;

/* Подготовка массива указателей на строки
 *  ptstr[0] --> строка 0
 *  ptstr[1] --> строка 1
 *  ptstr[2] --> строка 2
 *  и т.д.
 */
   ps= sbuf;
   for (ii=0; ii<Mask_Y; ++ii) {
      ptstr[ii]= ps;  ps+= Dx_source;
   }

   Yt= Yn_source;
```

```
   Yr= Yn_target+Y_centr;
   Sm= Mask_st[msknum];                 /* Шаг усреднения*/
   while (Yt <= Yk) {
      for (ii=0; ii<Mask_Y; ++ii)       /* Чтен исх строк */
         GetStr (ptstr[ii], Yt+ii, Xn_source, Xk_source);
      Xt=Xn_source;  pt= tbuf;
      while (Xt <= Xk) {
         pi= Maska; sr=0; sg=0; sb=0;   /* Суммированные RGB*/
         for (ii=0; ii<Mask_Y; ++ii) {
            ps= ptstr[ii] + (Xt-Xn_source);
            for (jj=0; jj<Mask_X; ++jj) {
               plut= &V_pal256[*ps++ & 255][0];
               s= *pi++;                /* Элемент маски */
               sr+= (s * *plut++);      /* Суммирование  */
               sg+= (s * *plut++);      /* по цветам с   */
               sb+= (s * *plut++);      /* весами маски  */
            }
         }
         sr /= Mask_sum; sg /= Mask_sum;  sb /= Mask_sum;
/* Поиск элемента ТЦ, наиболее подходящего для данных R,G,B */
         *pt++= V_clrint (sr, sg, sb);
         Xt+= Sm;
      }
      PutStr (tbuf,Yr++,                /* Запись строки */
              Xn_target+X_centr,
              Xn_target+X_centr + (--pt - tbuf));
      Yt+= Sm;
   }
   free (tbuf);
fr_sbuf:
   free (sbuf);
all:;
}  /* V_fltr3 */


/*================================================== T_FILTR.C
 *
 * ТЕСТ ФИЛЬТРАЦИИ
 *
 * Программа вначале строит два смещенных вектора
 * большими пикселами, затем последовательно для каждой
 * из пяти масок:
 * - фильтрует с непосредственным доступом к видеопамяти
 * - фильтрует с буферизацией растровых строк
 * - формирует усредненную картинку меньшего разрешения
 *   с непосредственным доступом к видеопамяти
```

```
 * - формирует усредненную картинку меньшего разрешения
 *   с буферизацией растровых строк
 *
 * После вывода очередной картинки ждет нажатия любой клавиши
 *
 * Виды масок:
 *   0: 1 1   1: 1 1 1   2: 1 1 1   3: 1 2 1
 *      1 1      1 1 1      1 2 1      2 4 2
 *               1 1 1      1 1 1      1 2 1
 *
 *  4: 1 1 1 1   5: 1 2  3  4  3 2 1
 *     1 1 1 1      2 4  6  8  6 4 2
 *     1 1 1 1      3 6  9 12  9 6 5
 *     1 1 1 1      4 8 12 16 12 8 4
 *                  3 6  9 12  9 6 5
 *                  2 4  6  8  6 4 2
 *                  1 2  3  4  3 2 1
 */

#include "V_VECTOR.C"
#include "VGA_256.C"
#include "V_FILTR.C"

#include <conio.h>
#include <graphics.h>
#include <stdio.h>

#define VECTOR 0  /* 0/1 - фикс вектор/ввод координат */

/*------------------------------------------------------ Grid
 * Строит сетку 10*10
 */

void Grid (void)
{  int Xn,Yn,Xk,Yk;
   setcolor (170);
   Xn= 0;  Xk= getmaxx();
   Yn= 0;  Yk= getmaxy();
   while (Xn <= Xk) {line (Xn,Yn,Xn,Yk); Xn+= 10; }
   Xn= 0;
   while (Yn <= Yk) {line (Xn,Yn,Xk,Yn); Yn+= 10; }
}  /* Grid */


/*------------------------------------------- main Filtr */
```

```
void main (void)
{
   int  ii, jj,
        mov_lin,                /* 0/1 - позиционир/отрезок */
        Xn,Yn,Xk,Yk,            /* Координаты отрезка       */
        fon=   140;             /* Индекс фона              */
   int  gdriver= DETECT, gmode;
   int  Xn_source, Yn_source,   /* Фильтруемая область      */
        Xk_source, Yk_source,
        Dx_source;
   int  Xn_target, Yn_target,   /* Результаты фильтрации    */
        Xk_target, Yk_target;
   int  msknum;                 /* Номер текущей маски  */
   char *ps;

   V_ini256 (&gdriver, &gmode, "");

   ps= (char *)V_pal256;
   for (ii=0; ii<=255; ++ii) {          /* Ч/б палитра    */
      jj= ii / 4;
      *ps++= jj; *ps++= jj; *ps++= jj;
      setrgbpalette (ii, jj, jj, jj);
   }
   setbkcolor(fon);                     /* Очистка экрана */
   cleardevice();
   Xk= getmaxx(); Yk= getmaxy();

/* Начальные установки для фильтрации */
   Xn_source= 0;                        /* Исходная область */
   Yn_source= 0;
   Xk_source= (Xk + 1)/2 - 1;
   Yk_source= Yk;
   Xn_target= Xk_source + 1;            /* Результ. область */
   Yn_target= 0;
   Xk_target= Xk;
   Yk_target= Yk_source;

   Dx_source= Xk_source-Xn_source+1;    /* X-размер исходной*/

#if VECTOR
   Grid ();
   mov_lin= 1;  Xn= 0;  Yn= 0;  Xk= 0;  Yk= 0;
   for (;;) {
      gotoxy (1, 1);
      printf("                                        \r");
```

```
      printf("mov_lin Xk Yk= (%d %d %d) ? ", mov_lin, Xk, Yk);
      scanf ("%d%d%d", &mov_lin, &Xk, &Yk);
      if (mov_lin < 0) cleardevice(); else
      if (!mov_lin) Grid (); else {
         if (mov_lin & 1) V_DDA (0, 0, Xk, Yk);
         if (mov_lin & 2) V_Bre (0, 0, Xk, Yk);
      }
   }
#else
   Xk= Dx_source / Pix_X - 1;
   Yk= (Yk_source-Yn_source+1) / Pix_Y - 1;
   V_DDA (Xn_source, Yn_source,    Xk, Yk-17);
   V_Bre (Xn_source, Yn_source+17, Xk, Yk);
   getch();
#endif

   ii= 0xF;     /* Обе фильтрации и оба сжатия */

   setfillstyle (SOLID_FILL, fon);

   for (msknum=0; msknum<6; ++msknum) {
      if (ii & 1) {     /* Фильтрация из видеоозу */
         bar (Xn_target, Yn_target, Xk_target, Yk_target);
         V_fltr0 (msknum,Xn_source,Yn_source,
                  Xk_source,Yk_source,Xn_target,Yn_target);
        getch ();
      }
      if (ii & 2) {     /* Фильтрация из буферов */
         bar (Xn_target, Yn_target, Xk_target, Yk_target);
         V_fltr1 (msknum,Xn_source,Yn_source,
                  Xk_source,Yk_source,Xn_target,Yn_target);
         getch ();
      }
      if (ii & 4) {     /* Сжатие из из видеоозу */
         bar (Xn_target, Yn_target, Xk_target, Yk_target);
         V_fltr2 (msknum,Xn_source,Yn_source,
                  Xk_source,Yk_source,Xn_target,Yn_target);
        getch ();
      }
      if (ii & 8) {     /* Сжатие из буферов */
         bar (Xn_target, Yn_target, Xk_target, Yk_target);
         V_fltr3 (msknum,Xn_source,Yn_source,
                  Xk_source,Yk_source,Xn_target,Yn_target);
         getch ();
      }
```

```
    }
    closegraph();
}   /* main */
```

0.15 Приложение 4. Процедуры генерации окружности

В данном приложении помещены процедуры генерации окружностей по алгоритму Брезенхема и Мичнера, а также программа T_Circle для тестирования данных процедур.

```
/*------------------------------------------------- V_Circle
 * Подпрограммы для генерации окружности
 * Pixel_circle - занесение пикселов с учетом симметрии
 * V_BRcirc     - генерирует окружность по алгоритму
 *                Брезенхема.
 * V_MIcirc     - генерирует окружность по алгоритму
 *                Мичнера.
 */

#include <graphics.h>

/*------------------------------------------- Pixel_circle
 * Заносит пикселы окружности по часовой стрелке
 */

static void Pixel_circle (xc, yc, x, y, pixel)
int  xc, yc, x, y, pixel;
{
   putpixel(xc+x, yc+y, pixel);
   putpixel(xc+y, yc+x, pixel);
   putpixel(xc+y, yc-x, pixel);
   putpixel(xc+x, yc-y, pixel);
   putpixel(xc-x, yc-y, pixel);
   putpixel(xc-y, yc-x, pixel);
   putpixel(xc-y, yc+x, pixel);
   putpixel(xc-x, yc+y, pixel);
}  /* Pixel_circle */


/*------------------------------------------------- V_BRcirc
 * Генерирует 1/8 окружности по алгоритму Брезенхема
 *
 * Процедура может строить 1/4 окружности.
 * Для этого надо цикл while заменить на for (;;)
 * и после Pixel_circle проверять достижение конца по условию
 * if (y <= end) break;
 * Где end устанавливается равным 0
 * В этом случае не нужен и последний оператор
 * if (x == y) Pixel_circle (xc, yc, x, y, pixel);
 * Генерацию 1/8 можно обеспечить задав end = r / sqrt (2)
 */
```

```
void V_BRcirc (xc, yc, r, pixel)
int  xc, yc, r, pixel;
{  int  x, y, z, Dd;
   x= 0;  y= r;  Dd= 2*(1-r);
   while (x < y) {
      Pixel_circle (xc, yc, x, y, pixel);
      if (!Dd) goto Pd;
      z= 2*Dd - 1;
      if (Dd > 0) {
         if (z + 2*x <= 0) goto Pd; else goto Pv;
      }
      if (z + 2*y > 0) goto Pd;
Pg:   ++x;      Dd= Dd + 2*x + 1;   continue; /* Горизонт */
Pd:   ++x; --y; Dd= Dd + 2*(x-y+1); continue; /* Диагонал */
Pv:   --y;      Dd= Dd - 2*y + 1;             /* Вертикал */
   }
   if (x == y) Pixel_circle (xc, yc, x, y, pixel);
}  /* V_BRcirc */


/*------------------------------------------------ V_MIcirc
 * Генерирует 1/8 окружности по алгоритму Мичнера
 */

void V_MIcirc (xc, yc, r, pixel)
int  xc, yc, r, pixel;
{  int  x, y, d;
   x= 0;  y= r;  d= 3 - 2*r;
   while (x < y) {
      Pixel_circle (xc, yc, x, y, pixel);
      if (d < 0) d= d + 4*x + 6; else {
         d= d + 4*(x-y) + 10;  --y;
      }
      ++x;
   }
   if (x == y) Pixel_circle (xc, yc, x, y, pixel);
}  /* V_MIcirc */




/*============================================== T_CIRCLE.C
 *
 * ТЕСТ ГЕНЕРАЦИИ ОКРУЖНОСТЕЙ
 *
```

```
 * Запрашивает ввод четырех чисел - координат центра,
 * радиуса и цвета построения: Xc Yc R Pix
 *
 * Затем строит заданную окружность по алгоритму Брезенхема
 * и концентрично с ней с радиусом, уменьшенным на 2, и
 * номером цвета, уменьшенным на 1, выдает окружность по
 * алгоритму Мичнера.
 *
 * При вводе Xc < 0 программа прекращает работу
 */

#include <graphics.h>
#include <stdio.h>
#include "V_CIRCLE.C"


/*------------------------------------------ MAIN T_CIRCLE.C
 */
void main (void)
{
   int   ii, Xc=300, Yc=240, R=238, Pix=14;
   int   gdriver = DETECT, gmode;

   initgraph(&gdriver, &gmode, "c:\tc\bgi");
   if ((ii= graphresult()) != grOk) {
      printf ("Err=%d\n", ii); goto all;
   }
   setbkcolor(0);
   cleardevice();

for (;;) {
gotoxy (1,1);
printf("                                                    \r");
printf("Xc, Yc, R, Pix= (%d %d %d %d) ? ", Xc,Yc,R,Pix);
scanf ("%d%d%d%d", &Xc, &Yc, &R, &Pix);
if (Xc < 0) break;
V_BRcirc (Xc, Yc, R, Pix);
V_MIcirc (Xc, Yc, R-2, Pix-1);
}
all:
   closegraph();
}
```

0.16 Приложение 5. Процедуры заполнения многоугольника

В данном приложении приведены две процедуры заливки многоугольника: V_FP0 и V_FP1. Обе они реализуют алгоритм построчного заполнения, описанный в разделе 5.

В данных процедурах все массивы используются, начиная с элемента с индексом 1, а не 0, как это принято в языке C.

0.16.1 V_FP0 — простая процедура заливки многоугольника

```
/*=======================================================  V_FP0
 * Простая (и не слишком эффективная) подпрограмма
 * однотонной заливки многоугольника методом построчного
 * сканирования. Имеет место дублирование закраски
 * строк.
 * Более эффективная программа, практически не дублирующая
 * занесение пикселов, - V_FP1 приведена далее в этом
 * приложении.
 */

#include <stdio.h>
#include <graphics.h>

#define MAXARR 300  /* Макс кол-во вершин многоугольника  */
#define MAXLST 300  /* Макс размер списка активных ребер  */


/*------------------------------------------------------ FILSTR
 * Заливает строку iy от ixn до ixk
 *
 * void FILSTR (int kod, int iy, int ixn, int ixk)
 */
void FILSTR (kod, iy, ixn, ixk)
int kod, iy, ixn, ixk;
{
   while (ixn <= ixk) putpixel (ixn++, iy, kod);
}  /* FILSTR */


/*-------------------------------------------------------   SORT
 * Сортирует n элементов iarr по возрастанию
 */
void SORT (n, iarr)
int  n, *iarr;
{  int  ii, jj, kk, ll, min;
   for (ii=0; ii<n; ++ii) {
```

```
      min= iarr[ll= ii];
      for (jj=ii+1; jj<n; ++jj)
         if ((kk= iarr[jj]) < min) {ll= jj; min= kk; }
      if (ll != ii) {iarr[ll]= iarr[ii]; iarr[ii]= min; }
   }
}  /* SORT */


/*--------------- Глобалы процедуры закраски ---------------*/

static int   KOD, NWER; /* Код заливки и количество вершин  */
static float *pt_X;     /* Массивы входных координат вершин */
static float *pt_Y;

static int   ntek;          /* Номер текущей вершины */

/* Список активных ребер */
static int   idlspi;        /* Длина списка активных ребер */
static int   IYREB[MAXLST]; /* Макс Y-коорд активных ребер */
static float RXREB[MAXLST]; /* Тек  X-коорд активных ребер */
static float RPRIR[MAXLST]; /* X-приращение на 1 шаг по Y  */


/*------------------------------------------------- OBRREB
 * По данным :
 *     NWER - количество вершин,
 *     ntek - номер текущей вершины,
 *     isd = -1/+1 - сдвиг для вычисления номера
 *           соседней вершины - слева/справа
 *     вычисляет DY,
 *     Если DY <  0 то вершина уже обработана,
 *     Если DY == 0 то вершины на одном Y, т.е.
 *                     строится горизонтальный отрезок,
 *     Если DY >  0 то формируется новый элемент списка
 *                     активных ребер
 */
static void OBRREB (isd)
int   isd;
{
   int   inext,iyt,ixt;
   float xt, xnext, dy;

   inext= ntek + isd;
   if (inext < 1) inext= NWER;
   if (inext > NWER) inext= 1;
```

```
   dy= pt_Y[inext] - pt_Y[ntek];
   if (dy < 0) goto RETOBR;
   xnext= pt_X[inext];
   xt= pt_X[ntek];
   if (dy != 0) goto DYNE0;
      iyt= pt_Y[ntek];
      inext= xnext;
      ixt= xt;
      FILSTR (KOD, iyt, inext, ixt);
      goto RETOBR;
DYNE0:
   idlspi++;
   IYREB[idlspi]= pt_Y[inext];
   RXREB[idlspi]= xt;
   RPRIR[idlspi]= (xnext - xt) / dy;
RETOBR:;
}  /* OBRREB */


/*-------------------------------------------------  V_FP0
 * Однотонно заливает многоугольник,
 * заданный координатами вершин
 *
 * void V_FP0 (int pixel, int kol, float *Px, float *Py)
 *
 */
void V_FP0 (pixel, kol, Px, Py)
int  pixel, kol;  float *Px, *Py;
{
int  ii, jj, kk;
int  iymin;         /* Мин  Y-координата многоугольн   */
int  iymax;         /* Макс Y-координата многоугольн   */
int  iysled;        /* Y-коорд появления новых вершин  */
int  iytek;
int  ikledg;        /* Кол-во вершин с данным iytek    */
int  ibgind;        /* Нач индекс таких вершин         */
int  iedg[MAXARR];  /* Y-коорд вершин по возрастанию   */
int  inom[MAXARR];  /* Их номера в исходном массиве Py */
int  irabx[MAXLST]; /* X-коорд пересечений в строке сканир */

   KOD= pixel;      /* Параметры в глобалы */
   NWER= kol;
   pt_X= Px;
   pt_Y= Py;
```

```
/* Построение массивов Y и их номеров */
   for (ii=1; ii<=kol; ++ii) {
      iedg[ii]= Py[ii]; inom[ii]= ii;
   }

/* Совместная сортировка Y-коорд вершин и их номеров */
   for (ii=1;  ii <= kol;  ++ii) {
      iymin= iedg[ii];
      ntek= ii;
      for (jj=ii+1;  jj <= kol;  ++jj)
         if (iedg[jj] < iymin) {iymin= iedg[jj]; ntek= jj; }
      if (ntek != ii) {
         iedg[ntek]= iedg[ii]; iedg[ii]= iymin;
         iymin= inom[ntek];
         inom[ntek]= inom[ii]; inom[ii]= iymin;
      }
   }

   idlspi= 0;                    /* Начальные присвоения */
   ibgind= 1;
   iytek= iedg[1];
   iymax= iedg[kol];

/* Цикл раскраски */

/* ikledg = кол-во вершин с данным iytek
 * ibgind = индексы таковых в массиве inom
 */
FORM_EDGES:
   ikledg= 0;
   for (ii=ibgind; ii<=kol; ++ii)
      if (iedg[ii] != iytek) break; else ikledg++;

/* Цикл построения списка активных ребер
 * и закрашивание горизонтальных ребер
 */

/* Построение списка активных ребер (CAP) */

   for (ii=1; ii<=ikledg; ++ii) {
      ntek= inom[ ibgind+ii-1];     /* Исх ном тек вершины */
      OBRREB (-1);                  /* DY с соседями затем */
      OBRREB (+1);                  /* либо отказ,  либо   */
                                    /* горизонталь, либо   */
   }                                /* измен списка активных*/
```

```
   if (!idlspi) goto KOHGFA;

   ii= ibgind + ikledg;         /* Y ближайшей вершины */
   iysled= iymax;
   if (ii < kol) iysled= iedg[ii];

/* Горизонтальная раскраска по списку */

   for (ii=iytek; ii<=iysled; ++ii) {
/* Выборка X-ов из списка активных ребер (CAP) */
      for (jj=1; jj <= idlspi; ++jj)
         irabx[jj]= RXREB[jj];
      SORT (idlspi, irabx+1);           /* Сортировка X-ов */
      for (jj=1; jj<=idlspi-1; jj+= 2)  /* Заливка */
         FILSTR (pixel, ii, irabx[jj], irabx[jj+1]);
      if (ii == iysled) continue;
      for (jj=1; jj <= idlspi; ++jj)    /* Перестройка CAP */
         RXREB[jj]= RXREB[jj] + RPRIR[jj];
   }

   if (iysled == iymax) goto KOHGFA;

/* Выбрасывание из списка всех ребер с YMAK ребра = YSLED */

   ii= 0;
M1:ii++;
M2:if (ii > idlspi) goto WYBROSILI;
      if (IYREB[ii] != iysled) goto M1;
         --idlspi;
         for (jj=ii;  jj <= idlspi;  ++jj) {
            IYREB[jj]= IYREB[kk= jj+1];
            RXREB[jj]= RXREB[kk];
            RPRIR[jj]= RPRIR[kk];
         }
         goto M2;
WYBROSILI:
   ibgind+= ikledg;
   iytek= iysled;

   goto FORM_EDGES;

KOHGFA:;
}  /* V_FP0 */
```

0.16.2 Тестовая процедуры V_FP0

```
/*------------------------------------------- main V_FP0 */

float Px[MAXARR] = {
   0.0,200.0,200.0,250.0,270.0,270.0,210.0,210.0,230.0,230.0
};
float Py[MAXARR] = {
   0.0,200.0,250.0,250.0,230.0,200.0,210.0,230.0,230.0,210.0
};

void main (void)
{
   int   ii, kol, grn, new, entry;
   int   gdriver = DETECT, gmode;

   kol= 5;              /* Кол-во вершин        */
   grn= 11;             /* Код пикселов границы */
   new= 14;             /* Код заливки          */
   entry= 1;

   initgraph(&gdriver, &gmode, "c:\tc\bgi");
   if ((ii= graphresult()) != grOk) {
      printf ("Err=%d\n", ii); goto all;
   }

m0:goto m2;
m1:++entry;
   printf("Vertexs, boundary_pixel, pixel= (%d %d %d) ? ",
            kol, grn, new);
   scanf ("%d%d%d", &kol, &grn, &new);
   if (kol < 0) goto all;

   for (ii=1; ii<=kol; ++ii) {
      printf ("Px[%d], Py[%d] = ? ", ii, ii);
      scanf  ("%d%d", &Px[ii], &Py[ii]);
   }

m2:
   setbkcolor(0);       /* Очистка экрана */
   cleardevice();

/* Заливка */
   V_FP0 (new, kol, Px, Py);

/* Построение границы */
```

```
   setcolor (grn);
   for (ii= 1; ii<kol; ++ii)
      line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
   line (Px[kol], Py[kol], Px[1], Py[1]);

/* При первом входе строится квадратик дырки */
   if (!entry) {
      for (ii=kol+1; ii<kol+4; ++ii)
         line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
      line (Px[kol+4], Py[kol+4], Px[kol+1], Py[kol+1]);
   }

   goto m1;

all:
   closegraph();
}
```

0.16.3 V_FP1 — эффективная процедура заливки многоугольника

```
/*=======================================================  V_FP1
 * Более эффективная по сравнению с V_FP0 подпрограмма
 * однотонной заливки многоугольника методом построчного
 * сканирования.
 *
 * Дублирование занесения пикселов практически отсутствует
 *
 */

#include <stdio.h>
#include <graphics.h>

#define MAXARR 300  /* Макс кол-во вершин многоугольника  */
#define MAXLST 300  /* Макс размер списка активных ребер  */


/*------------------------------------------------- FILSTR
 * Заливает строку iy от ixn до ixk
 *
 * void FILSTR (int kod, int iy, int ixn, int ixk)
 */
void FILSTR (kod, iy, ixn, ixk)
int kod, iy, ixn, ixk;
{
```

```
   while (ixn <= ixk) putpixel (ixn++, iy, kod);
}  /* FILSTR */



/*--------------- Глобалы процедуры закраски ---------------*/

static int   KOD, NWER; /* Код заливки и кол-во вершин      */
static float *pt_X;     /* Массивы входных координат вершин */
static float *pt_Y;

static int   IBGIND;        /* Номер след вершины в списке */
static int   IEDG[MAXARR];  /* Y-коорд вершин по возрастан */
static int   INOM[MAXARR];  /* и их номера в исх масс Py   */

/* Список активных ребер */
static int   IDLSPI;        /* Длина списка активных ребер */
static int   IYREB[MAXLST]; /* Макс Y-коорд активных ребер */
static float RXREB[MAXLST]; /* Тек  X-коорд активных ребер */
static float RPRIR[MAXLST]; /* X-приращение на 1 шаг по Y  */
static float RYSL[MAXLST];  /* Dy между тек и соседн верш  */
                            /* Dy <= 0.0 - обычная вершина */
                            /*     > 0.0 - локал экстремум */


/*------------------------------------------------- FORSPI
 * int  FORSPI (int IYBEG)
 *
 *  1) Формирует элементы списка для ребер,
 *     начинающихся в IYBEG;
 *  2) Вычисляет IBGIND - индекс начала следующей
 *     вершины в списке вершин;
 *  3) Возвращает IYSLED - Y координату ближайшей
 *     вершины, до которой можно заливать без
 *     перестройки списка.
 *
 *  Глобальные величины :
 *
 *  KOD    - код заливки
 *  NWER   - кол-во вершин в исходном многоугольнике,
 *  *pt_X  - X-координаты исходного многоугольника,
 *  *pt_Y  - Y-координаты исходного многоугольника,
 *  IEDG   - упорядоченный по возрастанию массив
 *           Y координат вершин исходного многоугольн
 *  INOM   - INOM[i] задает номер вершины в исходном
```

```
 *           многоугольнике для IEDG[i],
 *  IBGIND - индекс массивов IEDG, INOM
 *           определяет где может начаться ребро,
 *  IDLSPI - длина построенного списка активных ребер,
 *           состоящего из :
 *           IYREB  - макс координаты ребер,
 *           RXREB  - вначале мин, затем текущая X-координата,
 *           RPRIR  - приращение к X-координате на 1 шаг по Y,
 *           RYSL   - признак того что за вершина :
 *                    <= 0 - обычная,
 *                     > 0 - локальный экстремум
 *                     пересечение строки закраски
 *                     с экстремумом считается за 2 точки,
 *                     с обычной - за 1;
 */

static int  FORSPI (IYBEG)
int  IYBEG;
{

   int   i,ikledg,intek,intabs,isd;
   int   iyt,ixt,nrebra,inc,inpred,inposl;
   float xt, xc, yt, yc, dy;

/* ikledg = кол-во вершин с данным IYBEG */

   ikledg= 0;
   for (i=IBGIND; i<=NWER; ++i)
      if (IEDG[i] != IYBEG) break; else ++ikledg;

/* Цикл построения списка активных ребер
   и закрашивание горизонтальных ребер
 */

   for (i=1; i<=ikledg; ++i) {
/* Вычисл номера текущей вершины */
      intek= INOM[IBGIND+i-1];
      intabs= abs (intek);
      xt= pt_X[intabs];
      yt= pt_Y[intabs];

/*  Вычисл номеров предыд и послед вершин */
      if ((inpred= intabs - 1) < 1) inpred= NWER;
      if ((inposl= intabs + 1) > NWER) inposl= 1;
```

```
/*
 * По заданным :
 *    NWER   - кол-во вершин,
 *    intek  - номер текущей вершины,
 *    isd = 0/1 - правилу выбора соседней вершины -
 *                предыдущая/последующая
 *    вычисляет dy,
 *    Если dy <  0 то вершина уже обработана,
 *    Если dy == 0 то вершины на одном Y
 *                  При этом строится горизонтальный отрезок.
 *                  Факт закраски горизонтального ребра
 *                  отмечается отрицательным значением
 *                  соответствующего значения INOM.
 *    Если dy >  0 то формируется новый элемент списка
 *                  активных ребер
 */

      for (isd=0;  isd<=1; ++isd) {
         if (!isd) nrebra= inc= inpred; else {
            inc= inposl;  nrebra= intabs;
         }
         yc= pt_Y[inc];
         dy= yc - yt;
         if (dy < 0.0) continue;
         xc= pt_X[inc];
         if (dy != 0.0) goto DYNE0;
            if ((inc= INOM[nrebra]) < 0) continue;
            INOM[nrebra]= -inc;
            iyt= yt;
            inc= xc;
            ixt= xt;
            FILSTR (KOD, iyt, inc, ixt);
            continue;
DYNE0:   ++IDLSPI;
         IYREB[IDLSPI]= yc;
         RXREB[IDLSPI]= xt;
         RPRIR[IDLSPI]= (xc - xt) / dy;
         inc= (!isd) ? inposl : inpred;
         RYSL[IDLSPI]=  pt_Y[inc] - yt;
      }   /* цикла по isd */
   }  /* построения списка активных ребер */

/*  Вычисление Y ближайшей вершины */
   if ((i= (IBGIND += ikledg)) > NWER) i= NWER;
   return (IEDG[i]);
```

```
} /* Процедуры FORSPI */


/*-----------------------------------------------------  V_FP1
 * Однотонно заливает многоугольник,
 * заданный координатами вершин
 *
 * void V_FP1 (int pixel, int kol, float *Px, float *Py)
 *
 */
void V_FP1 (pixel, kol, Px, Py)
int  pixel, kol;  float *Px, *Py;
{
int  i,j,k,l;
int  iytek;    /* Y текущей строки сканирования        */
int  iymin;    /* Y-мин при сортировке массива Y-коорд */
int  iybeg;    /* Мин Y-координата заливки  */
int  iymak;    /* Max Y-координата заливки  */
int  iysled;   /* Y коорд ближайшей вершины, до которой */
               /* можно заливать без перестройки списка */
int  newysl;
int  ixmin;    /* X-мин при сортировке для тек строки */
int  ixtek;    /* X-тек при сортировке для тек строки */
int  irabx[MAXLST]; /* X-коорд пересечений в строке сканир */

   KOD= pixel;    /* Параметры в глобалы */
   NWER= kol;
   pt_X= Px;
   pt_Y= Py;

/*  Построение массивов Y и их номеров */
   for (i= 1; i<=NWER; ++i) {IEDG[i]= Py[i];  INOM[i]= i; }

/*  Совместная сортировка массивов IEDG, IHOM */
   for (i= 1; i<=NWER; ++i) {
      iymin= IEDG[i];
      k= 0;
      for (j=i+1; j<=NWER; ++j)
         if ((l= IEDG[j]) < iymin) {iymin= l; k= j; }
      if (k) {
         IEDG[k]= IEDG[i]; IEDG[i]= iymin;
         iymin= INOM[k];
         INOM[k]= INOM[i]; INOM[i]= iymin;
      }
   }
```

```
/* Начальные присвоения */
   IDLSPI= 0;
   IBGIND= 1;
   iybeg= IEDG[1];
   iymak= IEDG[NWER];

/* Формирование начального списка акт ребер */

   iysled= FORSPI (iybeg);
   if (!IDLSPI) goto KOHGFA;

/* Горизонтальная раскраска по списку */

ZALIWKA:

   for (iytek=iybeg; iytek<=iysled; ++iytek) {
      if (iytek == iysled) {    /* Y-координата перестройки */
         newysl= FORSPI (iytek);
         if (!IDLSPI) goto KOHGFA;
      }

/* Выборка и сортировка X-ов из списка ребер */
      l= 0;
      for (i=1; i<=IDLSPI; ++i)
         if (RYSL[i] > 0.0) irabx[++l]= RXREB[i];
         else RYSL[i]= 1.0;

      for (i=1;  i<=l; ++i) {
         ixmin= irabx[i];
         k= 0;
         for (j=i+1;  j<=l; ++j) {
            ixtek= irabx[j];
            if (ixtek < ixmin) {k= j; ixmin= ixtek; }
         }
         if (k) {irabx[k]= irabx[i];  irabx[i]= ixmin; }
      }  /* цикла сортировки */

/*  Собственно заливка */

      for (j=1;  j<=l-1;  j+= 2)
         FILSTR (KOD,iytek,irabx[j],irabx[j+1]);

      for (j=1;  j<=IDLSPI; ++j)        /*  Приращения X-ов */
         RXREB[j]= RXREB[j] + RPRIR[j];
   }  /* цикла горизонтальной раскраски */
```

```
   if (iysled == iymak) goto KOHGFA;

/*  Выбрасывание из списка всех ребер с YMAK ребра == YSLED */

   i= 0;
M1:++i;
M2:if (i > IDLSPI) goto WYBROSILI;
      if (IYREB[i] != iysled) goto M1;
         --IDLSPI;
         for (j=i;  j<=IDLSPI; ++j) {
            IYREB[j]= IYREB[k= j+1];
            RXREB[j]= RXREB[k];
            RPRIR[j]= RPRIR[k];
         }
         goto M2;
WYBROSILI:
   iybeg= iysled + 1;
   iysled= newysl;
   goto ZALIWKA;

KOHGFA:;
}  /* V_FP1 */
```

0.16.4 Тестовая процедуры V_FP1

```
/*---------------------------------------------- main V_FP1 */

float Px[MAXARR] = {
   0.0,200.0,200.0,250.0,270.0,270.0,210.0,210.0,230.0,230.0
};
float Py[MAXARR] = {
   0.0,200.0,250.0,250.0,230.0,200.0,210.0,230.0,230.0,210.0
};

void main (void)
{
   int   ii, kol, grn, new, entry;
   int   gdriver = DETECT, gmode;

   kol= 5;              /* Кол-во вершин        */
   grn= 11;             /* Код пикселов границы */
   new= 14;             /* Код заливки          */
   entry= 1;

   initgraph(&gdriver, &gmode, "c:\tc\bgi");
```

```
   if ((ii= graphresult()) != grOk) {
      printf ("Err=%d\n", ii); goto all;
   }

m0:goto m2;
m1:++entry;
   printf("Vertexs, boundary_pixel, pixel= (%d %d %d) ? ",
            kol, grn, new);
   scanf ("%d%d%d", &kol, &grn, &new);
   if (kol < 0) goto all;

   for (ii=1; ii<=kol; ++ii) {
      printf ("Px[%d], Py[%d] = ? ", ii, ii);
      scanf  ("%d%d", &Px[ii], &Py[ii]);
   }

m2:
   setbkcolor(0);       /* Очистка экрана */
   cleardevice();

/* Заливка */
   V_FP1 (new, kol, Px, Py);

/* Построение границы */
   setcolor (grn);
   for (ii= 1; ii<kol; ++ii)
      line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
   line (Px[kol], Py[kol], Px[1], Py[1]);

/* При первом входе строится квадратик дырки */
   if (!entry) {
      for (ii=kol+1; ii<kol+4; ++ii)
         line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
      line (Px[kol+4], Py[kol+4], Px[kol+1], Py[kol+1]);
   }

   goto m1;

all:
   closegraph();
}
```

0.17 Приложение 6. Процедуры заливки области

В данном приложении приведены три процедуры заливки гранично-определенной области с затравкой.

Первая процедура — V_FAB4R реализует рекурсивный алгоритм заполнения для 4-х связной области соответствующий алгоритму, помещенному в [4].

Вторая процедура — V_FAB4 реализует итеративный алгоритм заполнения для 4-х связной области близкий к алгоритму, помещенному в [3].

Характерная особенность таких алгоритмов — очень большие затраты памяти под рабочий стек и многократное дублирование занесения пикселов. Характерные значения для размера стека (см. ниже определение константы MAX_STK) около десяти тысяч байт при размере порядка 70×70 пикселов и очень сильно зависят от размеров заливаемой области, ее конфигурации и выбора начальной точки. Так, например, для заливки квадрата со стороной, равной 65 дискретам, и старте заливки из точки (20,20) относительно угла квадрата требуется 7938 байт для стека.

Третья процедура — V_FAST реализует алгоритм построчного заполнения с затравкой гранично-определенной области, близкий к соответствующему алгоритму из [3]. Отличительная черта таких алгоритмов — большие объемы программного кода, небольшие затраты памяти под рабочий стек и практически отсутствующее дублирование занесения пикселов. Характерные значения для размера стека (см. ниже определение константы MAX_STK) около сотни байт.

0.17.1 V_FAB4R — рекурсивная заливка 4-х связной области

```
/*------------------------------------------------- V_FAB4R
 * Подпрограммы для заливки с затравкой гранично-определенной
 * области 4-х связным алгоритмом:
 *
 * V_FAB4R  - заливка гранично-определенной
 *            области 4-х связным алгоритмом
 */

#include <graphics.h>
#include <stdio.h>

#define MAX_GOR 2048  /* Разрешение дисплея по X */
#define MAX_VER 2048  /* Разрешение дисплея по Y */

static int gor_max= MAX_GOR;
static int ver_max= MAX_VER;

/*------------------------------------------------- V_FAB4R
 * Заливка гранично-определенной области
 * 4-х связным алгоритмом
 */
void V_FAB4R (grn_pix, new_pix, x_isx, y_isx)
```

```
int grn_pix, new_pix, x_isx, y_isx;
{
   if (getpixel (x_isx, y_isx) != grn_pix &&
      getpixel (x_isx, y_isx) != new_pix)
   {
      putpixel (x_isx, y_isx, new_pix);
      V_FAB4R (grn_pix, new_pix, x_isx+1, y_isx);
      V_FAB4R (grn_pix, new_pix, x_isx,   y_isx+1);
      V_FAB4R (grn_pix, new_pix, x_isx-1, y_isx);
      V_FAB4R (grn_pix, new_pix, x_isx,   y_isx-1);
   }
}  /* V_FAB4 */
```

0.17.2 Тест процедуры V_FAB4R

```
/*------------------------------------------------- FAB4_MAIN
 */
void main (void)
{
   int   ii, kol, grn, new, entry;
   int   x_isx, y_isx;
   int   gdriver = DETECT, gmode;
   int   Px[256] = {200,200,250,270,270,210,210,230,230};
   int   Py[256] = {200,250,250,230,200,210,230,230,210};

   kol= 5;              /* Кол-во вершин        */
   grn= 11;             /* Код пикселов границы */
   new= 14;             /* Код заливки          */
   x_isx= 240;          /* Координаты затравки  */
   y_isx= 240;
   entry= 0;

   initgraph(&gdriver, &gmode, "c:\tc\bgi");
   if ((ii= graphresult()) != grOk) {
      printf ("Err=%d\n", ii); goto all;
   }

m0:goto m2;
m1:++entry;
   printf("Vertexs, boundary_pixel, new_pixel= (%d %d %d) ? ",
            kol, grn, new);
   scanf ("%d%d%d", &kol, &grn, &new);
   if (kol < 0) goto all;

   for (ii=0; ii<kol; ++ii) {
      printf ("Px[%d], Py[%d] = ? ", ii, ii);
```

```
      scanf  ("%d%d", &Px[ii], &Py[ii]);
   }

   printf ("X,Y isx= (%d %d) ? ", x_isx, y_isx);
   scanf ("%d%d", &x_isx, &y_isx);

m2:
   setbkcolor(0);
   cleardevice();

/* Построение границы */
   setcolor (grn);
   for (ii= 0; ii<kol-1; ++ii)
      line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
   line (Px[kol-1], Py[kol-1], Px[0], Py[0]);

/* При первом входе строится квадратик дырки */
   if (!entry) {
      for (ii= kol; ii<kol+3; ++ii)
         line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
      line (Px[kol+3], Py[kol+3], Px[kol], Py[kol]);
   }

/* Заливка */
   V_FAB4R (grn, new, x_isx, y_isx);
   goto m1;
all:
   closegraph();
}
```

0.17.3 V_FAB4 — итеративная заливка 4-х связной области

```
/*------------------------------------------------------ V_FAB4
 * Подпрограммы для заливки с затравкой гранично-определенной
 * области 4-х связным алгоритмом:
 *
 * Pop_Stk  - Локальная подпрограмма. Извлекает координаты
 *            пиксела из стека в глобальные скаляры xtek, ytek
 *
 * Push_Stk - Локальная подпрограмма. Заносит координаты
 *            пиксела в стек
 *
 * V_FAB4   - собственно заливка гранично-определенной
 *            области 4-х связным алгоритмом
```

```
 *
 * V_FA_SET - устанавливает количественные ограничения
 *            для заливки
 */

#include <alloc.h>
#include <graphics.h>
#include <stdio.h>

#define MAX_GOR 2048  /* Разрешение дисплея по X */
#define MAX_VER 2048  /* Разрешение дисплея по Y */
#define MAX_STK 8192  /* Размер стека координат заливки */

static int gor_max= MAX_GOR;
static int ver_max= MAX_VER;
static int stk_max= MAX_STK;
static int *pi_stk, *pn_stk;   /* Указ  стека заливки */
static int xtek, ytek;         /* Координаты из стека */
static int stklen;             /* Достигнутая глубина стека*/
                               /* только для отладочных    */
                               /* измерений программы      */

/*------------------------------------------------- Pop_Stk
 * Извлекает координаты пиксела из стека в xtek, ytek
 * Возвращает 0/1 - нет/есть ошибки
 */
static int  Pop_Stk ()
{  register int  otw;
   otw= 0;
   if (pi_stk <= pn_stk) ++otw; else {
      ytek= *--pi_stk;  xtek= *--pi_stk;
   }
   return (otw);
}  /* Pop_Stk */

/*------------------------------------------------- Push_Stk
 * Заносит координаты пиксела в стек
 * Возвращает -1/0 - нет места под стек/норма
 */
static int  Push_Stk (x, y)
register int x, y;
{
   register int glu;
   if ((glu= pi_stk - pn_stk) >= stk_max) x= -1; else {
      *pi_stk++= x;  *pi_stk++= y; x= 0;
```

```
      if (glu > stklen) stklen= glu;
   }
   return (x);
}  /* Push_Stk */


/*------------------------------------------------------ V_FAB4
 * Заливка гранично-определенной области
 * 4-х связным алгоритмом
 * Возвращает:
 * -1 - нет места под стек
 *  0 - норма
 */
int V_FAB4 (grn_pix, new_pix, x_isx, y_isx)
int grn_pix, new_pix, x_isx, y_isx;
{
   register int  pix, x, y, otw;

   otw= 0;

/* Инициализация стека */
   if ((pn_stk= (int *)malloc (stk_max)) == NULL) {
      --otw;  goto all;
   }
   pi_stk= pn_stk;

   Push_Stk (x_isx, y_isx);     /* Затравку в стек */

   while (pn_stk < pi_stk) {    /* Пока не исчерпан стек */
/* Выбираем пиксел из стека и красим его */
      Pop_Stk ();
      if (getpixel (x= xtek, y= ytek) != new_pix)
          putpixel (x, y, new_pix);

/* Проверяем соседние пикселы на необходимость закраски */
      if ((pix= getpixel (++x, y))   != new_pix &&
           pix != grn_pix) otw= Push_Stk (x, y);

      if ((pix= getpixel (--x, ++y)) != new_pix &&
           pix != grn_pix) otw= Push_Stk (x, y);

      if ((pix= getpixel (--x, --y)) != new_pix &&
           pix != grn_pix) otw= Push_Stk (x, y);

      if ((pix= getpixel (++x, --y)) != new_pix &&
           pix != grn_pix) otw= Push_Stk (x, y);
```

```
      if (otw) break;
   }
all:
   free (pn_stk);
   return (otw);
}  /* V_FAB4 */


/*------------------------------------------------- V_FA_SET
 * Устанавливает количественные ограничения для заливки
 */
void V_FA_SET (x_resolution, y_resolution, stack_length)
int  x_resolution, y_resolution, stack_length;
{
   if (x_resolution > 0 && x_resolution <= MAX_GOR)
      gor_max= x_resolution;
   if (y_resolution > 0 && y_resolution <= MAX_VER)
      ver_max= y_resolution;
/* Кол байт координат, заносимых в стек м.б. только четным */
   if (stack_length > 0) stk_max= stack_length & 0177776;
}  /* V_FA_SET */
```

0.17.4 Тест процедуры V_FAB4

```
/*------------------------------------------------- FAB4_MAIN
 */
void main (void)
{
   int   ii, kol, grn, new, entry;
   int   x_isx, y_isx;
   int   gdriver = DETECT, gmode;
   int   Px[256] = {200,200,250,270,270,210,210,230,230};
   int   Py[256] = {200,250,250,230,200,210,230,230,210};

   kol= 5;              /* Кол-во вершин        */
   grn= 11;             /* Код пикселов границы */
   new= 14;             /* Код заливки          */
   x_isx= 240;          /* Координаты затравки  */
   y_isx= 240;
   entry= 0;

   initgraph(&gdriver, &gmode, "c:\tc\bgi");
   if ((ii= graphresult()) != grOk) {
      printf ("Err=%d\n", ii); goto all;
```

```
   }

m0:goto m2;
m1:++entry;
   printf("Vertexs, boundary_pixel, new_pixel= (%d %d %d) ? ",
            kol, grn, new);
   scanf ("%d%d%d", &kol, &grn, &new);
   if (kol < 0) goto all;

   for (ii=0; ii<kol; ++ii) {
      printf ("Px[%d], Py[%d] = ? ", ii, ii);
      scanf  ("%d%d", &Px[ii], &Py[ii]);
   }

   printf ("X,Y isx= (%d %d) ? ", x_isx, y_isx);
   scanf ("%d%d", &x_isx, &y_isx);

m2:
   setbkcolor(0);
   cleardevice();

/* Построение границы */
   setcolor (grn);
   for (ii= 0; ii<kol-1; ++ii)
      line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
   line (Px[kol-1], Py[kol-1], Px[0], Py[0]);

/* При первом входе строится квадратик дырки */
   if (!entry) {
      for (ii= kol; ii<kol+3; ++ii)
         line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
      line (Px[kol+3], Py[kol+3], Px[kol], Py[kol]);
   }

/* Установка количественных ограничений для проц заливки */
   V_FA_SET (getmaxx()+1, getmaxy()+1, MAX_STK);

   stklen= 0;           /* Занятое кол-во байт в стеке */

/* Заливка */
   ii= V_FAB4 (grn, new, x_isx, y_isx);
   printf ("Answer= %d MaxStack=%d\n", ii, stklen);
   goto m1;

all:
```

```
   closegraph();
}
```

0.17.5 V_FAST — построчная заливка области

```
/*------------------------------------------------------ V_FAST
 * Подпрограммы заливки области с затравкой
 * построчным алгоритмом:
 *
 * Pop_Stk   - Локальная подпрограмма. Извлекает координаты
 *             пиксела из стека в глобальные скаляры xtek,ytek
 *
 * Push_Stk  - Локальная подпрограмма. Заносит координаты
 *             пиксела в стек
 *
 * Get_Video - Локальная подпрограмма. Читает строку из
 *             видеопамяти в глобальный буфер строки.
 *
 * Put_Video - Локальная подпрограмма. Копирует байты из
 *             глобального буфера строки в видеопамять.
 *
 * Search    - Локальная подпрограмма. Ищет затравочные
 *             пикселы в строке видеопамяти, находящейся
 *             в глобальном массиве.
 *
 * V_FAST    - Собственно подпрограмма построчной заливки
 *             гранично-определенной области
 *
 * V_FA_SET  - Устанавливает количественные ограничения
 *            для заливки
 */

#include <alloc.h>
#include <graphics.h>
#include <stdio.h>

#define MAX_GOR 2048  /* Разрешение дисплея по X */
#define MAX_VER 2048  /* Разрешение дисплея по Y */
#define MAX_STK 8192  /* Размер стека координат заливки */

static int gor_max= MAX_GOR;
static int ver_max= MAX_VER;
static int stk_max= MAX_STK;
static int *pi_stk, *pn_stk;   /* Указ  стека заливки */
static int xtek, ytek;         /* Координаты из стека */
static char *pc_video;         /* Указ на буфер строки */
```

```
static int stklen;              /* Достигнутая глубина стека*/
                                /* только для отладочных    */
                                /* измерений программы      */


/*------------------------------------------------- Pop_Stk
 * Извлекает координаты пиксела из стека в xtek, ytek
 * Возвращает 0/1 - нет/есть ошибки
 */
static int  Pop_Stk ()
{  register int  otw;
   otw= 0;
   if (pi_stk <= pn_stk) ++otw; else {
      ytek= *--pi_stk;  xtek= *--pi_stk;
   }
   return (otw);
}  /* Pop_Stk */

/*------------------------------------------------- Push_Stk
 * Заносит координаты пиксела в стек
 * Возвращает -1/0 - нет места под стек/норма
 */
static int  Push_Stk (x, y)
register int x, y;
{
   register int glu;
   if ((glu= pi_stk - pn_stk) >= stk_max) x= -1; else {
      *pi_stk++= x;  *pi_stk++= y; x= 0;
      if (glu > stklen) stklen= glu;
   }
   return (x);
}  /* Push_Stk */

/*------------------------------------------------- Get_Video
 * В байтовый буфер строки, заданный глобальным
 * указателем pc_video,
 * читает из видеопамяти пикселы y-строки от xbg до xen
 * Возвращает 0/1 - нет/есть ошибки
 */
static int  Get_Video (y, pcxbg, pcxen)
int y;  register char *pcxbg, *pcxen;
{  register int x;

   if (y>=0 && y<ver_max && pcxbg<=pcxen) {
      x= pcxbg - pc_video;
```

```
      do *pcxbg++= getpixel (x++, y); while (pcxbg <= pcxen);
      y= 0;
   } else y= 1;
   return (y);
}  /* Get_Video */

/*------------------------------------------------- Put_Video
 * Пикселы из буфера строки, начиная от указателя pxbg,
 * до указателя pxen пишет в y-строку видеопамяти
 * Возвращает 0/1 - нет/есть ошибки
 */
static int  Put_Video (y, pxbg, pxen)
int y;  register char *pxbg, *pxen;
{  register int  x;
   if (y>=0 && y<ver_max && pxbg<=pxen) {
      x= pxbg - pc_video;
      do putpixel (x++, y, *pxbg++); while (pxbg <= pxen);
      y= 0;
   } else y= 1;
   return (y);
}  /* Put_Video */

/*-------------------------------------------------- Search
 * Ищет затравочные пикселы в yt-строке видеопамяти,
 * находящейся по указателю pc_video, начиная от
 * указателя pcl до указателя pcr
 * grn - код граничного пиксела
 * new - код, которым перекрашивается область
 * Возвращает: 0/1 - не найден/найден затравочный
 */
static int  Search (yt, pcl, pcr, grn, new)
int yt;  char *pcl, *pcr;  int grn, new;
{  register int pix;
   register char *pc;
   int x, otw;

   otw= 0;
   while (pcl <= pcr) {
      pc= pcl;                          /* Указ тек пиксела */
/* Поиск крайнего правого не закрашенного пиксела в строке */
      while ((pix= *pc & 255) != grn && pix != new && pc<pcr)
         ++pc;

      if (pc != pcl) {          /* Найден закрашиваемый */
         ++otw;
```

```
         x= pc - pc_video;      /* Его координата в строке */
         if (pc != pcr || pix == grn || pix == new) --x;
         Push_Stk (x, yt);
      }
/* Продолжение анализа строки пока не достигнут прав пиксел */
      pcl= pc;
      while (((pix= *pc & 255) == grn || pix==new) && pc<pcr)
         ++pc;
      if (pc == pcl) ++pc;
      pcl= pc;
   }
   return (otw);
}  /* Search */


/*------------------------------------------------------ V_FAST
 * Построчная заливка с затравкой гранично-определенной
 * области
 *
 * int V_FAST (int grn_pix, int new_pix, int x_isx, int y_isx)
 *
 * Вход:
 * grn_pix - код граничного пиксела
 * new_pix - код заполняющего пиксела
 * x_isx   - координаты затравки
 * y_isx
 *
 * Возвращает:
 * -2 - нет места под растровую строку
 * -1 - нет места под стек
 *  0 - норма
 *  1 - при чтении пикселов из видеопамяти в буферную
 *      строки выход за пределы буферной строки
 *  2 - исчерпан стек при запросе координат пикселов
 *
 */
int V_FAST (grn_pix, new_pix, x_isx, y_isx)
int grn_pix, new_pix, x_isx, y_isx;
{
   register char *pcl;    /* Указ левого  пиксела в строке */
   register char *pcr;    /* Указ правого пиксела в строке */
   int  otw;

   otw= 0;

/* Инициализация стека */
```

```
   if ((pn_stk= (int *)malloc (stk_max)) == NULL) {
      --otw;  goto all;
   }
   pi_stk= pn_stk;

/* Заказ массива под растровую строку */
   if ((pc_video= malloc (gor_max)) == NULL) {
      otw= -2;  goto fre_stk;
   }

   Push_Stk (x_isx, y_isx);     /* Затравку в стек */

/* Цикл заливки строк до исчерпания стека */

   while (pi_stk > pn_stk) {

/* Запрос координат затравки из стека */
      if (Pop_Stk ()) {otw=2; break; }
      pcl= pcr= pc_video + xtek;   /* Указ затравки */

/* Запрос полной строки из видеопамяти */
      if (Get_Video (ytek, pc_video, pc_video+gor_max-1))
         {otw= 1;  break; }

/* Закраска затравки и вправо от нее */
      do *pcr++= new_pix; while ((*pcr & 255) != grn_pix);
      --pcr;                    /* Указ крайнего правого */

/* Закраска влево */
      while ((*--pcl & 255) != grn_pix) *pcl= new_pix;
      ++pcl;                    /* Указ крайнего левого */

/* Занесение подправленной строки в видеопамять */
      Put_Video (ytek, pcl, pcr);

/* Поиск затравок в строках ytek+1 и ytek-1,
 * начиная с левого подинтервала, заданного pcl, до
 * правого подинтервала, заданного pcr
 */
      if (!Get_Video (++ytek, pcl, pcr))
         Search (ytek, pcl, pcr, grn_pix, new_pix);

      if (!Get_Video (ytek-= 2, pcl, pcr))
         Search (ytek, pcl, pcr, grn_pix, new_pix);
   }
```

```
   free (pc_video);
fre_stk:
   free (pn_stk);
all:
   return (otw);
}  /* V_FAST */


/*------------------------------------------------- V_FA_SET
 * Устанавливает количественные ограничения для заливки
 */
void V_FA_SET (x_resolution, y_resolution, stack_length)
int  x_resolution, y_resolution, stack_length;
{
   if (x_resolution > 0 && x_resolution <= MAX_GOR)
      gor_max= x_resolution;
   if (y_resolution > 0 && y_resolution <= MAX_VER)
      ver_max= y_resolution;
/* Кол байт координат, заносимых в стек м.б. только четным */
   if (stack_length > 0) stk_max= stack_length & 0177776;
}  /* V_FA_SET */
```

0.17.6 Тест процедуры V_FAST

```
/*------------------------------------------------- FAST_MAIN
 */
void main (void)
{
   int   ii, kol, grn, new, entry;
   int   x_isx, y_isx;
   int   gdriver = DETECT, gmode;
   int   Px[256] = {200,200,250,270,270,210,210,230,230};
   int   Py[256] = {200,250,250,230,200,210,230,230,210};

   kol= 5;              /* Кол-во вершин        */
   grn= 11;             /* Код пикселов границы */
   new= 14;             /* Код заливки          */
   x_isx= 240;          /* Координаты затравки  */
   y_isx= 240;
   entry= 0;

   initgraph(&gdriver, &gmode, "c:\tc\bgi");
   if ((ii= graphresult()) != grOk) {
      printf ("Err=%d\n", ii); goto all;
   }
```

```
m0:goto m2;
m1:++entry;
   printf("Vertexs, boundary_pixel, new_pixel= (%d %d %d) ? ",
            kol, grn, new);
   scanf ("%d%d%d", &kol, &grn, &new);
   if (kol < 0) goto all;

   for (ii=0; ii<kol; ++ii) {
      printf ("Px[%d], Py[%d] = ? ", ii, ii);
      scanf  ("%d%d", &Px[ii], &Py[ii]);
   }

   printf ("X,Y isx= (%d %d) ? ", x_isx, y_isx);
   scanf ("%d%d", &x_isx, &y_isx);

m2:
   setbkcolor(0);
   cleardevice();

/* Построение границы */
   setcolor (grn);
   for (ii= 0; ii<kol-1; ++ii)
      line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
   line (Px[kol-1], Py[kol-1], Px[0], Py[0]);

/* При первом входе строится квадратик дырки */
   if (!entry) {
      for (ii= kol; ii<kol+3; ++ii)
         line (Px[ii], Py[ii], Px[ii+1], Py[ii+1]);
      line (Px[kol+3], Py[kol+3], Px[kol], Py[kol]);
   }

/* Установка количественных ограничений для проц заливки */
   V_FA_SET (getmaxx()+1, getmaxy()+1, MAX_STK);

   stklen= 0;           /* Занятое кол-во байт в стеке */

/* Заливка */
   ii= V_FAST (grn, new, x_isx, y_isx);
   printf ("Answer= %d MaxStack=%d\n", ii, stklen);
   goto m1;

all:
   closegraph();
```

```
}
```

0.18 Приложение 7. Процедуры отсечения отрезка

В данном приложении приведены процедуры, обеспечивающие выполнение отсечения по прямоугольному и многоугольному выпуклому окну и тестовая программа проверки работы процедур отсечения.

```
/*================================================== V_CLIP.C
 *
 * Подрограммы, связанные с отсечением:
 *
 * V_SetPclip - установить размеры многоугольного окна
 *              отсечения
 * V_SetRclip - установить размеры прямоугольного окна
 *              отсечения
 * V_GetRclip - опросить   размеры прямоугольного окна
 *              отсечения
 * V_CSclip   - отсечение по алгоритму Коэна-Сазерленда
 *              прямоугольное окно, кодирование
 *              концов отсекаемого отрезка
 * V_FCclip   - отсечение по алгоритму быстрого отсечения
 *              Алгоритм Собкова-Поспишила-Янга -
 *              прямоугольное окно, кодирование
 *              отсекаемого отрезка
 * V_LBclip   - отсечение по алгоритму Лианга-Барски
 *              прямоугольное окно, параметрическое
 *              представление линий
 * V_CBclip   - отсечение по алгоритму Кируса-Бека
 *              окно - выпуклый многоугольник,
 *              параметрическое представление линий
 */


/* Глобальные скаляры для алгоритмов отсечения по
 * прямоугольному окну - Коэна-Сазерленда, Fc-алгоритм,
 * Лианга-Барски
 */
static float Wxlef= 0.0,   /* Координаты левого нижнего и */
             Wybot= 0.0,   /* правого верхнего углов окна */
             Wxrig= 7.0,   /* отсечения                   */
             Wytop= 5.0;

/* Глобальные скаляры для алгоритма Кируса-Бека
 * отсечения по многоугольному окну
 */

/* Координаты прямоугольного окна */
static float Wxrect[4]= {0.0, 0.0, 7.0, 7.0 };
```

```
static float Wyrect[4]= {0.0, 5.0, 5.0, 0.0 };

/* Перепендикуляры к сторонам прямоугольного окна */
static float WxNrec[4]= {1.0,  0.0, -1.0, 0.0 };
static float WyNrec[4]= {0.0, -1.0,  0.0, 1.0 };

/* Данные для многоугольного окна */
static int   Windn=4;          /* Кол-во вершин у окна   */
static float *Windx=  Wxrect,  /* Координаты вершин окна */
             *Windy=  Wyrect;
static float *Wnormx= WxNrec,  /* Координаты нормалей    */
             *Wnormy= WyNrec;
```

0.18.1 V_SetPclip — установить многоугольник отсечения

```
/*------------------------------------------------ V_SetPclip
 * Устанавливает многоугольное окно отсечения
 * kv - количество вершин в окне
 * wx - X-координаты вершин
 * wy - Y-координаты вершин
 * nx - X-координаты нормалей к ребрам
 * ny - Y-координаты нормалей к ребрам
 *
 * Проверяет окно на выпуклость и невырожденность
 *
 * Если окно правильное, то
 * 1. Обновляет глобалы описания многоугольного окна:
 *    Windn=  kv;
 *    Windx=  wx;  Windy=  wy;  --Координаты вершин окна
 *    Wnormx= nx;  Wnormy= ny;  --Координаты перпендикуляров
 *
 * 2. Вычисляет координаты перепендикуляров к сторонам окна
 *
 * Возвращает:
 * 0 - норма
 * 1 - вершин менее трех
 * 2 - многоугольник вырожден в отрезок
 * 3 - многоугольник невыпуклый
 */

int  V_SetPclip (kv, wx, wy, nx, ny)
int  kv;  float *wx, *wy, *nx, *ny;
{  int   ii, jj, sminus, splus, szero, otw;
   float r,
         vox, voy,      /* Координаты (i-1)-й вершины */
         vix, viy,      /* Координаты i-й     вершины */
```

```
        vnx, vny;      /* Координаты (i+1)-й вершины */

/* Проверка на выпуклость
 * для этого вычисляются векторные произведения
 * смежных сторон и определяется знак
 * если все знаки == 0 то многоугольник вырожден
 * если все знаки >= 0 то многоугольник выпуклый
 * если все знаки <= 0 то многоугольник невыпуклый
 */
   otw= 0;
   if (--kv < 2) {++otw; goto all; }
   sminus= 0;
   splus=  0;
   szero=  0;
   vox= wx[kv];  voy= wy[kv];
   vix= *wx;     viy= *wy;
   ii= 0;
   do {
      if (++ii > kv) ii= 0;           /* Следующая  вершина */
      vnx= wx[ii];  vny= wy[ii];      /* Координаты (i+1)-й */
      r= (vix-vox)*(vny-viy) -        /* Вект произв ребер  */
         (viy-voy)*(vnx-vix);         /* смежных с i-й верш */
      if (r < 0) ++sminus; else
      if (r > 0) ++splus;  else ++szero;
      vox= vix;  voy= viy;            /* Обновлен координат */
      vix= vnx;  viy= vny;
   }  while (ii);

   if (!splus && !sminus)       /* Все векторные равны нулю */
      {otw= 2; goto all; }      /* Многоугольник вырожден   */
   if (splus && sminus)         /* Знакопеременн. векторные */
      {otw= 3; goto all; }      /* Многоугольник невыпуклый */

/* Установление глобалов для правильного окна */
   Windn= kv+1;                 /* Количество вершин у окна */
   Windx=  wx;  Windy=  wy;     /* Координаты вершин окна   */
   Wnormx= nx;  Wnormy= ny;     /* Координ. перпендикуляров */

/* Вычисление координат перпендикуляров к сторонам */

   vox= *wx; voy= *wy;
   ii= 0;
   do {
      if (++ii > kv) ii= 0;
      vix= wx[ii];  viy= wy[ii];     /* Текущая вершина */
```

```
      vnx= viy-voy; vny= vox-vix;    /* Поворот по часовой  */
      if (splus) {                   /* Внутр нормали влево */
         vnx= -vnx; vny= -vny;
      }
      *nx++= vnx;  *ny++= vny;
      vox= vix;  voy= viy;          /* Обновление координат */
   } while (ii);

all:
   return (otw);
}  /* V_SetPclip */
```

0.18.2 V_SetRclip — установить прямоугольник отсечения

```
/*------------------------------------------------ V_SetRclip
 * Устанавливает прямоугольное окно отсечения
 * Возвращает 0/1 - нет/есть ошибки в задании окна
 */
int  V_SetRclip (xleft, ybottom, xright, ytop)
float xleft, ybottom, xright, ytop;
{  int  otw;
   otw= 0;
   if (xleft >= xright || ybottom >= ytop) ++otw; else {
      Windn= 4;
      Windx= Wxrect;  Windy= Wyrect;         /* Вершины */
      Wxlef= Wxrect[0]= Wxrect[1]= xleft;
      Wybot= Wyrect[0]= Wyrect[3]= ybottom;
      Wxrig= Wxrect[2]= Wxrect[3]= xright;
      Wytop= Wyrect[1]= Wyrect[2]= ytop;
      Wnormx= WxNrec; Wnormy= WyNrec;        /* Нормали */
      WxNrec[0]=  1;  WyNrec[0]=  0;
      WxNrec[1]=  0;  WyNrec[1]= -1;
      WxNrec[2]= -1;  WyNrec[2]=  0;
      WxNrec[3]=  0;  WyNrec[3]=  1;
   }
   return (otw);
}  /* V_SetRclip */
```

0.18.3 V_GetRclip — опросить прямоугольник отсечения

```
/*------------------------------------------------ V_GetRclip
 * Возвращает текущее прямоугольное окно отсечения
 */
void V_GetRclip (xleft, ybottom, xright, ytop)
```

```
float *xleft, *ybottom, *xright, *ytop;
{
   *xleft=  Wxlef;  *ybottom= Wybot;
   *xright= Wxrig;  *ytop= Wytop;
}  /* V_GetRclip */
```

0.18.4 V_CSclip — отсечение Коэна-Сазерленда

```
/*------------------------------------------------- V_CSclip
 * Реализует алгоритм отсечения Коэна-Сазерленда с
 * кодированием концов отсекаемого отрезка
 *
 * int  V_CSclip (float *x0, float *y0, float *x1, float *y1)
 *
 * Отсекает отрезок, заданный значениями координат его
 * точек (x0,y0), (x1,y1), по окну отсечения, заданному
 * глобальными скалярами Wxlef, Wybot, Wxrig, Wytop
 *
 * Конечным точкам отрезка приписываются коды,
 * характеризующие его положение относительно окна отсечения
 * по правилу:
 *
 *  1001 | 1000 | 1010
 *  -----|------|-----
 *       | Окно |
 *  0001 | 0000 | 0010
 *  -----|------|-----
 *  0101 | 0100 | 0110
 *
 *  Отрезок целиком видим если оба его конца имеют коды 0000
 *  Если логическое И кодов концов не равно 0, то отрезок
 *  целиком вне окна и он просто отбрасывается.
 *  Если же результат этой операции = 0, то отрезок
 *  подозрительный. Он может быть и вне и пересекать окно.
 *  Для подозрительных отрезков определяются координаты их
 *  пересечений с теми сторонами, с которыми они могли бы
 *  пересечься в соответствии с кодами концов.
 *  При этом используется горизонтальность и вертикальность
 *  сторон окна, что позволяет определить одну из координат
 *  без вычислений.
 *  Часть отрезка, оставшаяся за окном отбрасывается.
 *  Оставшаяся часть отрезка проверяется на возможность его
 *  принятия или отбрасывания целиком. Если это невозможно,
 *  то процесс повторяется для другой стороны окна.
```

```
 *  На каждом цикле вычислений конечная точка отрезка,
 *  выходившая за окно, заменяется на точку, лежащую или на
 *  стороне окна или его продолжении.
 *
 *  Вспомогательная процедура Code вычисляет код положения
 *  для конца отрезка.
 *
 */


static float  CSxn, CSyn;   /* Координаты начала отрезка */

static int  CScode (void)  /* Определяет код точки xn, yn */
{  register int  i;
   i= 0;
   if (CSxn < Wxlef) ++i; else
   if (CSxn > Wxrig) i+= 2;
   if (CSyn < Wybot) i+= 4; else
   if (CSyn > Wytop) i+= 8;
   return (i);
}  /* CScode */


int   V_CSclip (x0, y0, x1, y1)
float *x0, *y0, *x1, *y1;
{
   float  CSxk, CSyk;   /* Координаты конца отрезка  */
   int    cn, ck,       /* Коды концов отрезка */
          visible,      /* 0/1 - не видим/видим*/
          ii, s;        /* Рабочие переменные  */
   float  dx, dy,       /* Приращения координат*/
          dxdy,dydx,    /* Наклоны отрезка к сторонам */
          r;            /* Рабочая переменная  */

   CSxk= *x1; CSyk= *y1;
   CSxn= *x1; CSyn= *y1; ck= CScode ();
   CSxn= *x0; CSyn= *y0; cn= CScode ();

/* Определение приращений координат и наклонов отрезка
 * к осям. Заодно сразу на построение передается отрезок,
 * состоящий из единственной точки, попавшей в окно
 */
   dx= CSxk - CSxn;
   dy= CSyk - CSyn;
   if (dx != 0) dydx= dy / dx; else {
      if (dy == 0) {
```

```
         if (cn==0 && ck==0) goto out; else goto all;
      }
   }
   if (dy != 0) dxdy= dx / dy;

/* Основной цикл отсечения */
   visible= 0;  ii= 4;
   do {
      if (cn & ck) break;       /* Целиком вне окна    */
      if (cn == 0 && ck == 0) { /* Целиком внутри окна */
         ++visible;  break;
      }
      if (!cn) {                /* Если Pn внутри окна, то */
         s= cn; cn= ck; ck= s;  /* перестить точки Pn,Pk и */
         r=CSxn; CSxn=CSxk; CSxk=r;  /* их коды, чтобы Pn  */
         r=CSyn; CSyn=CSyk; CSyk=r;  /* оказалась вне окна */
      }
      /* Теперь отрезок разделяется. Pn помещается в точку
       * пересечения отрезка со стороной окна.
       */
      if (cn & 1) {         /* Пересечение с левой стороной */
         CSyn= CSyn + dydx * (Wxlef-CSxn);
         CSxn= Wxlef;
      } else if (cn & 2) {  /* Пересечение с правой стороной*/
         CSyn= CSyn + dydx * (Wxrig-CSxn);
         CSxn= Wxrig;
      } else if (cn & 4) {  /* Пересечение в нижней стороной*/
         CSxn= CSxn + dxdy * (Wybot-CSyn);
         CSyn= Wybot;
      } else if (cn & 8) {  /*Пересечение с верхней стороной*/
         CSxn= CSxn + dxdy * (Wytop-CSyn);
         CSyn= Wytop;
      }
      cn= CScode ();        /* Перевычисление кода точки Pn */
   } while (--ii >= 0);
   if (visible) {
out:  *x0= CSxn;  *y0= CSyn;
      *x1= CSxk;  *y1= CSyk;
   }
all:
   return (visible);
}  /* V_CSclip */
```

0.18.5 V_FCclip — Fast Clipping-алгоритм

```
/*------------------------------------------------ V_FCclip
 *  Реализует алгоритм отсечения FC (Fast Clipping)
 *  Собкова-Поспишила-Янга, с кодированием линий
 *
 * int  V_FCclip (float *x0, float *y0, float *x1, float *y1)
 *
 * Отсекает отрезок, заданный значениями координат его
 * точек (x0,y0), (x1,y1), по окну отсечения, заданному
 * глобальными скалярами Wxlef, Wybot, Wxrig, Wytop
 *
 * Возвращает:
 * -1 - ошибка в задании окна
 *  0 - отрезок не видим
 *  1 - отрезок видим
 */


static float FC_xn, FC_yn, FC_xk, FC_yk;

static void Clip0_Top(void)
{FC_xn= FC_xn + (FC_xk-FC_xn)*(Wytop-FC_yn)/(FC_yk-FC_yn);
 FC_yn= Wytop; }

static void Clip0_Bottom(void)
{FC_xn= FC_xn + (FC_xk-FC_xn)*(Wybot-FC_yn)/(FC_yk-FC_yn);
 FC_yn= Wybot; }

static void Clip0_Right(void)
{FC_yn= FC_yn + (FC_yk-FC_yn)*(Wxrig-FC_xn)/(FC_xk-FC_xn);
 FC_xn= Wxrig; }

static void Clip0_Left(void)
{FC_yn= FC_yn + (FC_yk-FC_yn)*(Wxlef-FC_xn)/(FC_xk-FC_xn);
 FC_xn= Wxlef; }

static void Clip1_Top(void)
{FC_xk= FC_xk + (FC_xn-FC_xk)*(Wytop-FC_yk)/(FC_yn-FC_yk);
 FC_yk= Wytop; }

static void Clip1_Bottom(void)
{FC_xk= FC_xk + (FC_xn-FC_xk)*(Wybot-FC_yk)/(FC_yn-FC_yk);
 FC_yk= Wybot; }

static void Clip1_Right(void)
```

```
{FC_yk= FC_yk + (FC_yn-FC_yk)*(Wxrig-FC_xk)/(FC_xn-FC_xk);
 FC_xk= Wxrig; }

static void Clip1_Left(void)
{FC_yk= FC_yk + (FC_yn-FC_yk)*(Wxlef-FC_xk)/(FC_xn-FC_xk);
 FC_xk= Wxlef; }


int  V_FCclip (x0, y0, x1, y1)
float *x0, *y0, *x1, *y1;
{  int  Code= 0;
   int  visible= 0;             /* Отрезок невидим */

   FC_xn= *x0;  FC_yn= *y0;
   FC_xk= *x1;  FC_yk= *y1;

/*
 * Вычисление значения Code - кода отрезка
 * Биты 0-3 - для конечной точки, 4-7 - для начальной
 *
 */
   if (FC_yk > Wytop) Code+= 8; else
   if (FC_yk < Wybot) Code+= 4;

   if (FC_xk > Wxrig) Code+= 2; else
   if (FC_xk < Wxlef) Code+= 1;

   if (FC_yn > Wytop) Code+= 128; else
   if (FC_yn < Wybot) Code+= 64;

   if (FC_xn > Wxrig) Code+= 32; else
   if (FC_xn < Wxlef) Code+= 16;

/* Отсечение для каждого из 81-го случаев */

   switch (Code) {

     /* Из центра */

     case 0x00: ++visible;  break;
     case 0x01: Clip1_Left() ;   ++visible;  break;
     case 0x02: Clip1_Right();  ++visible;  break;
     case 0x04: Clip1_Bottom(); ++visible;  break;
     case 0x05: Clip1_Left() ;
                if (FC_yk < Wybot) Clip1_Bottom();
                ++visible;  break;
```

```
case 0x06: Clip1_Right();
           if (FC_yk < Wybot) Clip1_Bottom();
           ++visible;  break;
case 0x08: Clip1_Top();    ++visible;  break;
case 0x09: Clip1_Left() ;
           if (FC_yk > Wytop) Clip1_Top();
           ++visible;  break;
case 0x0A: Clip1_Right();
           if (FC_yk > Wytop) Clip1_Top();
           ++visible;  break;


/* Слева */

case 0x10: Clip0_Left();   ++visible;
case 0x11: break;                          /* Отброшен */
case 0x12: Clip0_Left();   Clip1_Right();
           ++visible;  break;
case 0x14: Clip0_Left();
           if (FC_yn < Wybot) break;       /* Отброшен */
           Clip1_Bottom();
           ++visible;
case 0x15: break;                          /* Отброшен */
case 0x16: Clip0_Left();
           if (FC_yn < Wybot) break;       /* Отброшен */
           Clip1_Bottom();
           if (FC_xk > Wxrig) Clip1_Right();
           ++visible;
           break;
case 0x18: Clip0_Left();
           if (FC_yn > Wytop) break;       /* Отброшен */
           Clip1_Top();
           ++visible;
case 0x19: break;                          /* Отброшен */
case 0x1A: Clip0_Left();
           if (FC_yn > Wytop) break;       /* Отброшен */
           Clip1_Top();
           if (FC_xk > Wxrig) Clip1_Right();
           ++visible;
           break;


/* Справа */

case 0x20: Clip0_Right(); ++visible;  break;
case 0x21: Clip0_Right(); Clip1_Left(); ++visible;
```

```
case 0x22: break;                          /* Отброшен */
case 0x24: Clip0_Right();
           if (FC_yn < Wybot) break;       /* Отброшен */
           Clip1_Bottom();
           ++visible;
           break;
case 0x25: Clip0_Right();
           if (FC_yn < Wybot) break;       /* Отброшен */
           Clip1_Bottom();
           if (FC_xk < Wxlef) Clip1_Left();
           ++visible;
case 0x26: break;                          /* Отброшен */
case 0x28: Clip0_Right();
           if (FC_yn > Wytop) break;       /* Отброшен */
           Clip1_Top();
           ++visible;
           break;
case 0x29: Clip0_Right();
           if (FC_yn > Wytop) break;       /* Отброшен */
           Clip1_Top();
           if (FC_xk < Wxlef) Clip1_Left();
           ++visible;
case 0x2A: break;                          /* Отброшен */


/* Снизу */

case 0x40: Clip0_Bottom(); ++visible;  break;
case 0x41: Clip0_Bottom();
           if (FC_xn < Wxlef) break;       /* Отброшен */
           Clip1_Left() ;
           if (FC_yk < Wybot) Clip1_Bottom();
           ++visible;
           break;
case 0x42: Clip0_Bottom();
           if (FC_xn > Wxrig) break;       /* Отброшен */
           Clip1_Right();
           ++visible;
case 0x44:
case 0x45:
case 0x46: break;                          /* Отброшен */
case 0x48: Clip0_Bottom();
           Clip1_Top();
           ++visible;
           break;
case 0x49: Clip0_Bottom();
```

```
            if (FC_xn < Wxlef) break;       /* Отброшен */
            Clip1_Left() ;
            if (FC_yk > Wytop) Clip1_Top();
            ++visible;
            break;
case 0x4A: Clip0_Bottom();
            if (FC_xn > Wxrig) break;       /* Отброшен */
            Clip1_Right();
            if (FC_yk > Wytop) Clip1_Top();
            ++visible;
            break;


/* Снизу слева */

case 0x50: Clip0_Left();
            if (FC_yn < Wybot) Clip0_Bottom();
            ++visible;
case 0x51: break;                           /* Отброшен */
case 0x52: Clip1_Right();
            if (FC_yk < Wybot) break;       /* Отброшен */
            Clip0_Bottom();
            if (FC_xn < Wxlef) Clip0_Left();
            ++visible;
case 0x54:
case 0x55:
case 0x56: break;                           /* Отброшен */
case 0x58: Clip1_Top();
            if (FC_xk < Wxlef) break;       /* Отброшен */
            Clip0_Bottom();
            if (FC_xn < Wxlef) Clip0_Left();
            ++visible;
case 0x59: break;                           /* Отброшен */
case 0x5A: Clip0_Left();
            if (FC_yn > Wytop) break;       /* Отброшен */
            Clip1_Right();
            if (FC_yk < Wybot) break;       /* Отброшен */
            if (FC_yn < Wybot) Clip0_Bottom();
            if (FC_yk > Wytop) Clip1_Top();
            ++visible;
            break;


/* Снизу-справа */

case 0x60: Clip0_Right();
```

```
            if (FC_yn < Wybot) Clip0_Bottom();
            ++visible;
            break;
case 0x61: Clip1_Left() ;
            if (FC_yk < Wybot) break;       /* Отброшен */
            Clip0_Bottom();
            if (FC_xn > Wxrig) Clip0_Right();
            ++visible;
case 0x62:
case 0x64:
case 0x65:
case 0x66: break;                           /* Отброшен */
case 0x68: Clip1_Top();
            if (FC_xk > Wxrig) break;       /* Отброшен */
            Clip0_Right();
            if (FC_yn < Wybot) Clip0_Bottom();
            ++visible;
            break;
case 0x69: Clip1_Left() ;
            if (FC_yk < Wybot) break;       /* Отброшен */
            Clip0_Right();
            if (FC_yn > Wytop) break;       /* Отброшен */
            if (FC_yk > Wytop) Clip1_Top();
            if (FC_yn < Wybot) Clip0_Bottom();
            ++visible;
case 0x6A: break;                           /* Отброшен */


/* Сверху */

case 0x80: Clip0_Top();
            ++visible;
            break;
case 0x81: Clip0_Top();
            if (FC_xn < Wxlef) break;       /* Отброшен */
            Clip1_Left() ;
            ++visible;
            break;
case 0x82: Clip0_Top();
            if (FC_xn > Wxrig) break;       /* Отброшен */
            Clip1_Right();
            ++visible;
            break;
case 0x84: Clip0_Top();
            Clip1_Bottom();
            ++visible;
```

```
            break;
case 0x85: Clip0_Top();
            if (FC_xn < Wxlef) break;       /* Отброшен */
            Clip1_Left() ;
            if (FC_yk < Wybot) Clip1_Bottom();
            ++visible;
            break;
case 0x86: Clip0_Top();
            if (FC_xn > Wxrig) break;       /* Отброшен */
            Clip1_Right();
            if (FC_yk < Wybot) Clip1_Bottom();
            ++visible;
case 0x88:
case 0x89:
case 0x8A: break;                           /* Отброшен */


/* Сверху-слева */

case 0x90: Clip0_Left();
            if (FC_yn > Wytop) Clip0_Top();
            ++visible;
case 0x91: break;                           /* Отброшен */
case 0x92: Clip1_Right();
            if (FC_yk > Wytop) break;       /* Отброшен */
            Clip0_Top();
            if (FC_xn < Wxlef) Clip0_Left();
            ++visible;
            break;
case 0x94: Clip1_Bottom();
            if (FC_xk < Wxlef) break;       /* Отброшен */
            Clip0_Left();
            if (FC_yn > Wytop) Clip0_Top();
            ++visible;
case 0x95: break;                           /* Отброшен */
case 0x96: Clip0_Left();
            if (FC_yn < Wybot) break;       /* Отброшен */
            Clip1_Right();
            if (FC_yk > Wytop) break;       /* Отброшен */
            if (FC_yn > Wytop) Clip0_Top();
            if (FC_yk < Wybot) Clip1_Bottom();
            ++visible;
case 0x98:
case 0x99:
case 0x9A: break;                           /* Отброшен */
```

```
    /* Сверху-справа */

    case 0xA0: Clip0_Right();
               if (FC_yn > Wytop) Clip0_Top();
               ++visible;
               break;
    case 0xA1: Clip1_Left() ;
               if (FC_yk > Wytop) break;       /* Отброшен */
               Clip0_Top();
               if (FC_xn > Wxrig) Clip0_Right();
               ++visible;
    case 0xA2: break;                          /* Отброшен */
    case 0xA4: Clip1_Bottom();
               if (FC_xk > Wxrig) break;       /* Отброшен */
               Clip0_Right();
               if (FC_yn > Wytop) Clip0_Top();
               ++visible;
               break;
    case 0xA5: Clip1_Left() ;
               if (FC_yk > Wytop) break;       /* Отброшен */
               Clip0_Right();
               if (FC_yn < Wybot) break;       /* Отброшен */
               if (FC_yk < Wybot) Clip1_Bottom();
               if (FC_yn > Wytop) Clip0_Top();
               ++visible;
    case 0xA6:                                 /* Отброшен */
    case 0xA8:
    case 0xA9:
    case 0xAA: break;


    /* Ошибка */

    default:   visible= -1;
               break;
  }  /* switch */

  if (visible > 0) {
     *x0= FC_xn;  *y0= FC_yn;
     *x1= FC_xk;  *y1= FC_yk;
  }
  return (visible);
}  /* V_FCclip */
```

0.18.6 V_LBclip — алгоритм Лианга-Барски

```
/*------------------------------------------------ V_LBclip
 *  Реализует алгоритм отсечения Лианга-Барски
 *  с параметрическим заданием линий
 *
 * int  V_LBclip (float *x0, float *y0, float *x1, float *y1)
 *
 * Отсекает отрезок, заданный значениями координат его
 * точек (x0,y0), (x1,y1), по окну отсечения, заданному
 * глобальными скалярами Wxlef, Wybot, Wxrig, Wytop
 *
 * Возвращает:
 *  0 - отрезок не видим
 *  1 - отрезок видим
 */

static float LB_t0, LB_t1;

static int  LB_tclip (p, q)
float p, q;
{
   int   accept;
   float r;

   accept= 1;                          /* Отрезок принят */
   if (p == 0) {
      if (q < 0) accept= 0;            /* Отбрасывание */
   } else {
      r= q/p;
      if (p < 0) {
         if (r > LB_t1) accept= 0;     /* Отбрасывание */
         else if (r > LB_t0) LB_t0= r;
      } else {
         if (r < LB_t0) accept= 0;     /* Отбрасывание */
         else if (r < LB_t1) LB_t1= r;
      }
   }
   return (accept);
}  /* LB_tclip */


int  V_LBclip (x0, y0, x1, y1)
float *x0, *y0, *x1, *y1;
{  int   visible;
   float dx, dy;
```

```
   visible= 0;
   LB_t0= 0;  LB_t1= 1;
   dx= *x1 - *x0;
   if (LB_tclip (-dx, *x0-Wxlef)) {
      if (LB_tclip (dx, Wxrig-*x0)) {
         dy= *y1 - *y0;
         if (LB_tclip (-dy, *y0-Wybot)) {
            if (LB_tclip (dy, Wytop-*y0)) {
               if (LB_t1 < 1) {
                  *x1= *x0 + LB_t1*dx;
                  *y1= *y0 + LB_t1*dy;
               }
               if (LB_t0 > 0) {
                  *x0= *x0 + LB_t0*dx;
                  *y0= *y0 + LB_t0*dy;
               }
               ++visible;
            }
         }
      }
   }
   return (visible);
}  /* V_LBclip */
```

0.18.7 V_CBclip — алгоритм Кируса-Бека

```
/*------------------------------------------------- V_CBclip
 *  Реализует алгоритм отсечения Кируса-Бека
 *  по произвольному выпуклому многоугольнику
 *  с параметрическим заданием линий
 *
 * int  V_CBclip (float *x0, float *y0, float *x1, float *y1)
 *
 * Отсекает отрезок, заданный значениями координат его
 * точек (x0,y0), (x1,y1), по окну отсечения, заданному
 * глобальными скалярами:
 * int   Windn - количество вершин в окне отсечения
 * float *Windx, *Windy   - массивы X,Y координат вершин
 * float *Wnormx, *Wnormy - массивы координат нормалей
 *                          к ребрам
 *
 * Возвращает:
 *  0 - отрезок не видим
```

```
 *  1 - отрезок видим
 */


int  V_CBclip (x0, y0, x1, y1)
float *x0, *y0, *x1, *y1;
{  int   ii, jj, visible, kw;
   float xn, yn, dx, dy, r;
   float CB_t0, CB_t1;          /* Параметры концов отрезка */
   float Qx, Qy;                /* Положение относ ребра */
   float Nx, Ny;                /* Перпендикуляр к ребру */
   float Pn, Qn;                /**/

   kw= Windn - 1;               /* Ребер в окне */
   visible= 1;
   CB_t0= 0;  CB_t1= 1;
   dx= *x1 - (xn= *x0);
   dy= *y1 - (yn= *y0);

   for (ii=0; ii<=kw; ++ii) {   /* Цикл по ребрам окна */
      Qx= xn - Windx[ii];       /* Положения относ ребра */
      Qy= yn - Windy[ii];
      Nx= Wnormx[ii];           /* Перепендикуляр к ребру */
      Ny= Wnormy[ii];
      Pn= dx*Nx + dy*Ny;        /* Скалярные произведения */
      Qn= Qx*Nx + Qy*Ny;

/* Анализ расположения */
      if (Pn == 0) {            /* Паралл ребру или точка */
         if (Qn < 0) {visible= 0;  break; }
      } else {
         r= -Qn/Pn;
         if (Pn < 0) {          /* Поиск верхнего предела t */
            if (r < CB_t0) {visible= 0;  break; }
            if (r < CB_t1) CB_t1= r;
         } else {               /* Поиск нижнего предела t */
            if (r > CB_t1) {visible= 0;  break; }
            if (r > CB_t0) CB_t0= r;
         }
      }
   }
   if (visible) {
      if (CB_t0 > CB_t1) visible= 0; else {
         if (CB_t0 > 0) {
            *x0= xn + CB_t0*dx;
            *y0= yn + CB_t0*dy;
```

```
         }
         if (CB_t1 < 1) {
            *x1= xn + CB_t1*dx;
            *y1= yn + CB_t1*dy;
         }
      }
   }
   return (visible);
}  /* V_CBclip */
```

0.18.8 Тест процедур отсечения

```
/*================================================== T_CLIP.C
 *
 * ТЕСТ ПРОЦЕДУР ОТСЕЧЕНИЯ
 */

#include <time.h>
#include <stdio.h>

/*-------------------------------------------------- V_DMclip
 *  Пустышка для процедур отсечения
 */

int  V_DMclip (x0, y0, x1, y1)
float *x0, *y0, *x1, *y1;
{  int   visible;
   visible= 1;
   return (visible);
}  /* V_DMclip */



/*-------------------------------------------------- ClipMsg
 * Печатает сообщение о результатах отсечения
 */
void ClipMsg (proc, visible, x0, y0, x1, y1, dt)
char *proc; int visible; float x0, y0, x1, y1, dt;
{
   if (visible < 0) {
      printf("*** ERROR (%s LineClip) - ", proc);
      printf("ошибка в координатах окна. ");
      printf("Прерывание с кодом ошибки 1.");
      exit (1);
   } else if (visible == 0)
```

```
      printf ("%s: Line is no visible dt=%f\n", proc, dt);
   else
      printf ("%s: ClipLine: x0=%f y0=%f x1=%f y1=%f dt=%f\n",
               proc, x0, y0, x1, y1, dt);
}  /* ClipMsg */


/*------------------------------------------- MAIN T_CLIP.C
 */
void main (void)
{
   float Wxn, Wyn, Wxk, Wyk;
   float Xn, Yn, Xk, Yk, x0, y0, x1, y1;
   int   ii, numb= 1;
   float X_wind[100], Y_wind[100];
   float X_norm[100], Y_norm[100];
   int  visible;
   float dt;
   time_t t1, t2;
   long  ll, powt=10l;

   if (numb) goto set_win;

m0:printf ("----Вершин= %d ? ", numb);
   scanf  ("%d", &numb);
   for (ii=0; ii<numb; ++ii) {
      printf ("X_wind[%d], Y_wind[%d] ? ", ii, ii);
      scanf  ("%f%f", &X_wind[ii], &Y_wind[ii]);
   }
   ii= V_SetPclip (numb, X_wind, Y_wind, X_norm, Y_norm);
   printf ("V_SetPclip= %d\n", ii);
   if (ii) goto m0;
   for (ii=0; ii<numb; ++ii)
      printf ("ind=%d X_norm=%f, Y_norm=%f\n",
               ii, X_norm[ii], Y_norm[ii]);
   if (ii) goto m0;


/* Задание окна отсечения */
set_win:
   powt= 1l;
   V_GetRclip (&Wxn, &Wyn, &Wxk, &Wyk);
   for (;;) {
      printf ("Window: (Xn=%f Yn=%f Xk=%f Yk=%f) ? ",
               Wxn, Wyn, Wxk, Wyk);
      scanf  ("%f%f%f%f", &Wxn, &Wyn, &Wxk, &Wyk);
```

```
      if (!V_SetRclip (Wxn, Wyn, Wxk, Wyk)) break;
      printf ("Error in a window boundarys\n");
   }

/* Ввод координат отрезка */
   Xn= Wxn-1.0;  Yn= Wyn-1.0;  Xk= Wxk+1.0;  Yk= Wyk+1.0;

   for (;;) {
      printf ("------------- ");
      printf ("ClipWindow: Xn=%f Yn=%f Xk=%f Yk=%f\n",
               Wxlef, Wybot, Wxrig, Wytop);
      printf ("New Line: (Xn=%f Yn=%f Xk=%f Yk=%f) ? ",
               Xn, Yn, Xk, Yk);
      scanf  ("%f%f%f%f", &Xn, &Yn, &Xk, &Yk);

      ll= powt;
      t1= time(NULL);
      do {
         x0= Xn; y0= Yn; x1= Xk; y1= Yk;
         visible= V_DMclip (&x0, &y0, &x1, &y1);
      } while (--ll > 0l);
      t2= time (NULL);
      dt= ((float)(t2 - t1));
      ClipMsg ("DM", visible, x0, y0, x1, y1, dt);


      ll= powt;
      t1= time(NULL);
      do {
         x0= Xn; y0= Yn; x1= Xk; y1= Yk;
         visible= V_CSclip (&x0, &y0, &x1, &y1);
      } while (--ll > 0l);
      t2= time (NULL);
      dt= ((float)(t2 - t1));
      ClipMsg ("CS", visible, x0, y0, x1, y1, dt);

      ll= powt;
      t1= time(NULL);
      do {
         x0= Xn; y0= Yn; x1= Xk; y1= Yk;
         visible= V_FCclip (&x0, &y0, &x1, &y1);
      } while (--ll > 0l);
      t2= time (NULL);
      dt= ((float)(t2 - t1));
      ClipMsg ("FC", visible, x0, y0, x1, y1, dt);
```

```
      ll= powt;
      t1= time(NULL);
      do {
         x0= Xn; y0= Yn; x1= Xk; y1= Yk;
         visible= V_LBclip (&x0, &y0, &x1, &y1);
      } while (--ll > 0l);
      t2= time (NULL);
      dt= ((float)(t2 - t1));
      ClipMsg ("LB", visible, x0, y0, x1, y1, dt);

      ll= powt;
      t1= time(NULL);
      do {
         x0= Xn; y0= Yn; x1= Xk; y1= Yk;
         visible= V_CBclip (&x0, &y0, &x1, &y1);
      } while (--ll > 0l);
      t2= time (NULL);
      dt= ((float)(t2 - t1));
      ClipMsg ("CB", visible, x0, y0, x1, y1, dt);
   }
}
```

0.19 Приложение 8. Процедуры отсечения многоугольника

В данном приложении содержатся процедуры V_Plclip, реализующие простой алгоритм отсечения произвольного многоугольника по выпуклому многоугольному окну отсечения и тестовая программа проверки их работы.

Процедуры реализуют алгоритм, который, как и алгоритм Сазерленда-Ходгмана, последовательно отсекает весь многоугольник по каждому из ребер окна отсечения.

0.19.1 V_Plclip — простой алгоритм отсечения многоугольника

```
/*================================================= V_PLCLIP
 * Файл V_PLCLIP.C - процедуры простого алгоритма отсечния
 *                   многоугольника
 * Последняя редакция:
 * 25.12.93  17:25
 */

#include <graphics.h>
#include "V_WINDOW.C"   /* Глобалы и проц работы с окнами */

/* Включаемый файл V_WINDOW.C содержит
 * подрограммы и глобалы для окон:
 *
 * V_SetPclip - установить размеры многоугольного окна
 *              отсечения
 * V_GetPclip - опросить параметры многоугольного окна
 *              отсечения
 * V_SetRclip - установить размеры прямоугольного окна
 *              отсечения
 * V_GetRclip - опросить   размеры прямоугольного окна
 *              отсечения
 *
 * Глобальные скаляры для алгоритмов отсечения по
 * прямоугольному окну - Коэна-Сазерленда, Fc-алгоритм,
 * Лианга-Барски
 *
 * static float Wxlef= 100.0, -- Координаты левого нижнего и
 *              Wybot= 100.0, -- правого верхнего углов окна
 *              Wxrig= 300.0, -- отсечения
 *              Wytop= 200.0;
 *
 * Глобальные скаляры для алгоритма Кируса-Бека
 * отсечения по многоугольному окну
 *
 * Координаты прямоугольного окна
```

```
 * static float Wxrect[4]= {100.0, 100.0, 300.0, 300.0 };
 * static float Wyrect[4]= {100.0, 200.0, 200.0, 100.0 };
 *
 * Перепендикуляры к сторонам прямоугольного окна
 * static float WxNrec[4]= {1.0,  0.0, -1.0, 0.0 };
 * static float WyNrec[4]= {0.0, -1.0,  0.0, 1.0 };
 *
 * Данные для многоугольного окна
 * static int   Windn=4;          -- Кол-во вершин у окна
 * static float *Windx=  Wxrect,  -- Координаты вершин окна
 *              *Windy=  Wyrect;
 * static float *Wnormx= WxNrec,  -- Координаты нормалей
 *              *Wnormy= WyNrec;
 */


/*------------------------------------------------ Pl_clip0
 * Служебная процедура, отсекает многоугольник
 * относительно одного ребра окна
 *
 * Отсечение выполняется в цикле по сторонам многоугольника
 * При первом входе в цикл только вычисляются величины для
 * начальной точки: координаты, скалярное произведение,
 * определяющее ее расположение относительно ребра окна, и
 * код расположения.
 * При последующих входах в цикл эти значения вычисляются
 * для очередной вершины.
 * По значениям кодов расположения вершин для стороны
 * многоугольника определяется как она расположена
 * относительно ребра и вычисляются координаты результирующего
 * многоугольника.
 */

static int Pl_clip0 (N_reb, N_in, X_in, Y_in, X_ou, Y_ou)
int   N_reb, N_in;
float *X_in, *Y_in, *X_ou, *Y_ou;
{
   int  ii, jj;
   int  pozb,      /* Коды расположения начальной точки  */
        pozn,      /* многоугольника и точек тек стороны */
        pozk;      /* 0/1/2 - пред точка вне/на/внутри   */
   float Rx,Ry;    /* Координаты начала ребра окна */
   float Nx, Ny;   /* Нормаль к  ребру окна */
   float xb, yb;   /* Начальная точка многоугольника  */
   float xn, yn;   /* Начальная точка текущей стороны */
   float xk, yk;   /* Конечная  точка текущей стороны */
```

```
   float t;        /* Значение параметра точки пересечения */
   float Qb,Qn,Qk; /* Скалярные произведения */
   float *ptx_ou;

/* Запрос параметров ребра окна */
   Rx= Windx[N_reb];  Ry= Windy[N_reb];
   Nx= Wnormx[N_reb]; Ny= Wnormy[N_reb];

/* Цикл отсчения многоугольника ребром окна */
   ii= 0;  ++N_in;  ptx_ou= X_ou;
   while (--N_in >= 0) {
      if (N_in) {
         xk= *X_in++;  yk= *Y_in++;   /* Кон точка стороны  */
         Qk= (xk-Rx)*Nx + (yk-Ry)*Ny; /* Параметр положения */
         pozk= 1;                     /* 1 - точка на гр. */
         if (Qk < 0) --pozk; else     /* 0 - точка вне    */
         if (Qk > 0) ++pozk;          /* 2 - точка внутри */
      } else {
         xk= xb;  yk= yb;  Qk= Qb;  pozk= pozb;
      }
      if (!ii) {
         xb= xn= xk; yb= yn= yk; Qb= Qn= Qk; pozb= pozn= pozk;
         ++ii; continue;
      }
      jj= 0;
      switch (pozn*3 + pozk) {     /* Стар Нов Выход     */
         case 0: goto no_point;    /*  вне-вне нет       */
         case 1: goto endpoint;    /*  вне-на  конечная  */
         case 2: goto intersec;    /*  вне-вну перес,кон */
         case 3: goto no_point;    /*  на -вне нет       */
         case 4:                   /*  на -на  конечная  */
         case 5: goto endpoint;    /*  на -вну конечная  */
         case 6: goto no_end;      /*  вну-вне пересечен */
         case 7:                   /*  вну-на  конечная  */
         case 8: goto endpoint;    /*  вну-вну конечная  */
      }
no_end: ++jj;
intersec:
      t= Qn/(Qn-Qk);
      *X_ou++= xn + t*(xk-xn);
      *Y_ou++= yn + t*(yk-yn);
      if (!jj) {
endpoint:
         *X_ou++= xk;  *Y_ou++= yk;
      }
```

```
no_point:
      xn= xk;  yn= yk;  Qn= Qk;  pozn= pozk;
   }
   return (X_ou - ptx_ou);
}  /* Pl_clip0 */



/*------------------------------------------------- V_Plclip
 * Простая процедура отсечения многоугольника
 * N_in       - число вершин во входном многоугольнике
 * X_in, Y_in - координаты вершин отсекаемого мног-ка
 *              этот массив будет испорчен
 * Возвращает:
 *  < 0 - ошибки
 * >= 0 - количество вершин в выходном многоугольнике
 * X_ou, Y_ou - координаты вершин отсеченного многоугольника
 */

int  V_Plclip (N_in, X_in, Y_in, X_ou, Y_ou)
int   N_in;
float *X_in, *Y_in, *X_ou, *Y_ou;
{
   int  ii, N_ou; float *ptr;

   if ((N_ou= N_in) < 3) {N_ou= -1;  goto all; }
   if (Windn < 3) {N_ou= -2;  goto all; }
   for (ii=0; ii<Windn; ++ii) {
      N_ou= Pl_clip0 (ii, N_ou, X_in, Y_in, X_ou, Y_ou);
      ptr= X_in;  X_in= X_ou;  X_ou= ptr;
      ptr= Y_in;  Y_in= Y_ou;  Y_ou= ptr;
   }
   if (!(Windn & 1)) {
      ii= N_ou;
      while (--ii >= 0) {*X_ou++= *X_in++; *Y_ou++= *Y_in++; }
   }
all:
   return N_ou;
}  /* V_Plclip */
```

0.19.2 Тест процедуры V_Plclip

```
/*================================================= T_PLCLIP
 * ТЕСТ процедуры V_Plclip для отсечения многоугольника
 *
```

```
 * При первом входе устанавливается восьмиугольное окно
 * отсечения:
 * X: 150, 100, 100, 150, 250, 300, 300, 250
 * Y: 100, 150, 250, 300, 300, 250, 150, 100
 *
 * И на нем отсекается треугольник:
 * (10,160),(90,220),(170,160)
 *
 * Затем выдается запрос на количество вершин в
 * новом отсекаемом многоугольнике:
 * --- Vertexs in polyline= XX ?
 * При вводе числа > 0 будут запрашиваться координаты вершин
 *                     много-ка и выполняться его отсечение
 * При вводе числа = 0 программа затребует ввод координат
 *                     нового прямоугольного окна отсечения
 * При вводе числа < 0 программа запросит переустановку
 *                     многоугольного окна отсечения:
 *
 * ----Vertexs in clip window= XX ?
 * При вводе числа > 0 будут запрашиваться координаты вершин.
 * При вводе числа = 0 программа затребует ввод координат
 *                     прямоугольного окна.
 * При вводе числа < 0 программа закончится.
 */


#include <graphics.h>
#include "V_PLCLIP.C"

/*------------------------------------------------- DrawPoly
 * Чертит контур многоугольника
 */
void DrawPoly (col, n, x, y)
int col, n; float *x, *y;
{  int  ii, jj;
   setcolor (col);
   for (ii=0; ii<n; ++ii) {
      if ((jj= ii+1) >= n) jj= 0;
      line (x[ii], y[ii], x[jj], y[jj]);
   }
}  /* DrawPoly */

/*------------------------------------------------- CLR_STR
 * Зачищает строку выводом в нее пробелов
 */
void CLR_STR (void)
```

```
{
printf ("                                                    \r");
}

/*---------------------------------------------- PLCLIP_MAIN
 */
void main (void)
{  int   ii, jj,
         fon;           /* Индекс фона  */
   float Wxn,Wyn,       /* Прямоугольный отсекатель */
         Wxk,Wyk;
   int   N_wind= 0;     /* Вводимый отсекатель */
   float X_wind[100],
         Y_wind[100];
   float X_norm[100],
         Y_norm[100];
   int   wnum;          /* Запрошенный отсекатель */
   float *wx,*wy,*wnx,*wny;
   int   N_poly= 0;     /* Отсекаемый многугольник */
   float X_poly[100],
         Y_poly[100];
   int   N_clip= 0;     /* Отсеченный многугольник */
   float X_clip[100],
         Y_clip[100];
   int   entry= 0;      /* 0/1 - нет/был вывод по умолчанию */
   int   gdriver= DETECT, gmode;

   initgraph (&gdriver, &gmode, "");
   fon= 0;                      /* Цвет фона */

   setbkcolor(fon);             /* Очистка экрана */
   cleardevice();

/*--------------- Установить окно отсечения ----------------*/
new_window:
   gotoxy (1,1);
   if (!entry) {
      N_wind= 8;  wx= X_wind;  wy= Y_wind;
      *wx++= 150; *wx++= 100;  *wx++= 100; *wx++= 150;
      *wy++= 100; *wy++= 150;  *wy++= 250; *wy++= 300;

      *wx++= 250; *wx++= 300;  *wx++= 300; *wx++= 250;
      *wy++= 300; *wy++= 250;  *wy++= 150; *wy++= 100;
      goto wr_window;
   }
```

```
   if (!N_poly) goto set_rect;


/*---------- Задание многоугольного окна отсечения ---------*/
set_window:
   CLR_STR ();
   printf ("----Vertexs in clip window= %d ? ", N_wind);
   scanf  ("%d", &N_wind);
   if (N_wind < 0) goto all;
   if (!N_wind) goto set_rect;
   for (ii=0; ii<N_wind; ++ii) {
      CLR_STR ();
      printf ("X_wind[%d], Y_wind[%d] ? ", ii, ii);
      scanf  ("%f%f", &X_wind[ii], &Y_wind[ii]);
   }
wr_window:
   jj= V_SetPclip (N_wind, X_wind, Y_wind, X_norm, Y_norm);
   if (jj) {
      printf ("Error=%d in polyline window\n", jj);
      goto set_window;
   } else goto ou_win;


/*---------- Задание прямоугольного окна отсечения ---------*/
set_rect:
   V_GetRclip (&Wxn, &Wyn, &Wxk, &Wyk);
get_rect:
   CLR_STR ();
   printf ("Rect window: (Xn=%f Yn=%f Xk=%f Yk=%f) ? ",
            Wxn, Wyn, Wxk, Wyk);
   scanf  ("%f%f%f%f", &Wxn, &Wyn, &Wxk, &Wyk);
wr_rect:
   jj= V_SetRclip (Wxn, Wyn, Wxk, Wyk);
   if (jj) {
      printf ("Error=%d in rectangle window\n", jj);
      goto get_rect;
   }


/*--------------- Прорисовка окна отсечения ----------------*/
ou_win:
   wnum= V_GetPclip (&wx, &wy, &wnx, &wny);
   DrawPoly (LIGHTRED, wnum, wx, wy);
```

```
/*------- Ввод координат отсекаемого многоугольника --------*/

set_poly:
   gotoxy (1,1);
   if (!entry) { /* При первом входе отрисовка по умолчанию */
      N_poly= 3;
      X_poly[0]=  10;  X_poly[1]=  90;  X_poly[2]= 170;
      Y_poly[0]= 160;  Y_poly[1]= 220;  Y_poly[2]= 160;
   } else {
      CLR_STR ();
      printf ("--- Vertexs in polyline= %d ? ",N_poly);
      scanf  ("%d", &N_poly);
      if (N_poly <= 0) goto new_window;
      for (ii=0; ii<N_poly; ++ii) {
         printf ("                                      \r");
         printf ("X_poly[%d], Y_poly[%d] ? ", ii, ii);
         scanf  ("%f%f", &X_poly[ii], &Y_poly[ii]);
      }
   }
   ++entry;

/*---------- Прорисовка отсекателя и отсекаемого -----------*/
   wnum= V_GetPclip (&wx, &wy, &wnx, &wny);
   DrawPoly (LIGHTRED, wnum, wx, wy);
   DrawPoly (LIGHTGREEN, N_poly, X_poly, Y_poly);


/*----------------------- Отсечение ------------------------*/
   N_clip= V_Plclip(N_poly, X_poly, Y_poly, X_clip, Y_clip);

/*----------------- Прорисовка отсеченного -----------------*/
   DrawPoly (YELLOW, N_clip, X_clip, Y_clip);
   goto set_poly;

all:
   closegraph();
}  /* PLCLIP_MAIN */
```

Пётр Вильгельмович Вельтмандер

МАШИННАЯ ГРАФИКА
(Учебное пособие в 3-х книгах)

Книга 2

ОСНОВНЫЕ АЛГОРИТМЫ

Подписано в печать	27 мая 1997 г.	Формат 60 × 84/16
Офсетная печать.		Уч.-изд. л. 15
Заказ	Тираж 120 экз.	Цена 2000

Редакционно-издательский отдел Новосибирского университета;
участок оперативной полиграфии НГУ; 630090, Новосибирск 90,
ул. Пирогова, 2.